UNIVERSIDAD AUTONOMA
DE MADRID

Escuela Politécnica Superior
Departamento de Ingeniería Informática

TESIS DOCTORAL

EVOLUCIÓN GRAMATICAL Y SEMÁNTICA

Autor:
Marina de la Cruz Echeandía

Directores:
Dr. Manuel Alfonseca
Dr. Alfonso Ortega

Madrid, junio de 2010

A Nicolás.

AGRADECIMIENTOS

Por fin llegó este momento tan deseado. Llegué al final de un camino, que para mí, ha sido duro. Pero no he estado sola. A todas las personas que han estado a mi lado les quiero dar las gracias.

Gracias a mis directores de tesis, Manuel Alfonseca y Alfonso Ortega. Siempre tengo cosas que aprender de vosotros.

Gracias a mis compañeros de trabajo. Me habéis alegrado el camino. En especial, me siento en deuda con Javier. Gracias, sin tu apoyo no hubiera terminado este trabajo.

Gracias a Irene. Has hecho que LaTeX fuera divertido.

Gracias a mis padres. Siempre me acompañáis en todos los caminos que comienzo, algunos buenos, y otros no.

Gracias a mi hijo Nicolás. Sé que me estás esperando para jugar.

Y sobre todo, gracias a tí, compañero de viaje. Nadie sabe mejor que tú cómo ha sido el camino. Gracias por quererme, mimarme, enseñarme, ayudarme ... Te quiero.

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Motivación y contribuciones

La programación genética propuesta por Koza a principios de la década de los noventa es un algoritmo evolutivo que permite generar automáticamente programas en el lenguaje de programación LISP. Desde que Koza hizo su propuesta original hasta la actualidad ha habido un importante desarrollo en el área de la programación genética. Han sido propuestas distintas variantes del método de manera que, en la actualidad, están admitidas varias clasificaciones de los mismos. La más extendida, clasifica los métodos en función de la representación de los programas que forman la población objeto del proceso evolutivo. Por una parte, la *programación genética basada en árboles* agrupa a todos los métodos que representan los programas en forma de árbol, y al que pertenece la programación genética original de Koza. Otro importante grupo es el de la *programación genética lineal* en la que los programas se representan como secuencias de instrucciones de lenguajes imperativos, o código máquina, o también cadenas numéricas. Uno de los problemas que tienen que resolver los sistemas de programación genética es asegurar la clausura sintáctica, es decir, asegurar la corrección sintáctica de los programas que forman la población. Cada sistema resuelve este problema de una manera particular. En concreto, hay un grupo de propuestas que utilizan gramáticas independientes del contexto como formalismo para asegurar la clausura sintáctica del método. A este grupo se le denomina *programación genética gramatical* o *programación genética basada en gramáticas*. Las distintas clasificaciones de los métodos de programación genética permiten que haya métodos que pertenezcan a más de un grupo. Este es el caso de la *Evolución Gramatical*, pertenece al grupo de la programación genética lineal porque representa los programas como

una secuencia de bits, y también pertenece a la categoría de programación genética gramatical porque utiliza una gramática independiente del contexto para garantizar la clausura sintáctica. Los autores de la Evolución Gramatical, Michael O'Neill y Conor Ryan, definen su método como "algoritmo evolutivo capaz de generar automáticamente programas escritos en cualquier lenguaje de programación".

La motivación de esta tesis es intentar responder a la pregunta: ¿Qué resultados se obtendrían incorporando en la programación genética lo que podríamos llamar clausura semántica o corrección semántica de los programas que forman la población? ¿Mejoraría el rendimiento si los programas además de ser sintácticamente correctos también lo son semánticamente?. Para intentar responder a las preguntas anteriores, se proponen en esta tesis dos métodos que extienden con semántica la Evolución Gramatical, es decir, que garantizan la corrección semántica de los programas. El primer método utiliza las gramáticas de atributos como formalismo para la representación del lenguaje que genera las soluciones, y el segundo hace uso de las gramáticas de Christiansen.

Junto con los métodos teóricos propuestos en esta tesis, se ha desarrollado una aplicación, **GEEMMA** , con el objetivo de disponer de un entorno en el que probar dichos métodos y así poder validar su funcionamiento y evaluar su rendimiento.

Una de las posibles aplicaciones de las técnicas que se proponen en esta tesis es la programación automática de sistemas complejos, como por ejemplo los sistemas de computación con membranas (sistemas P) o los sistemas de *splicing* (sistemas H). Entendemos por programación automática de un sistema complejo la obtención de un diseño de dicho sistema de manera que se comporte de acuerdo a una funcionalidad previamente establecida. La programación automática de sistemas complejos mediante las técnicas propuestas en esta tesis supone, en primer lugar, definir el lenguaje con el que se van a expresar los sistemas. Esa definición en algunos casos se hará con gramáticas de atributos, y en otros con gramáticas de Christiansen. En segundo lugar, es necesario programar un simulador capaz de ejecutar los sistemas programados y así poder evaluar el comportamiento del sistema. Una vez que se dispone del lenguaje generador de los sistemas y de un evaluador de los mismos, es ya posible ejecutar el algoritmo evolutivo para encontrar el sistema que mejor se adapte a la funcionalidad previamente establecida. Si los sistemas sólo se definen sintácticamente, como lo hace la técnica de Evolución Gramatical, el espacio de búsqueda es mucho mayor que cuando

se incorporan aspectos semánticos como lo hacen las técnicas propuestas en esta tesis.

1.2. Estructura de la tesis

La estructura de este documento es la siguiente. En el capítulo 2 se contextualiza el trabajo desarrollado en esta tesis. El estudio se centra en dos líneas principales: por una parte, se describe el área de la programación genética ya que en ella se ubica la Evolución Gramatical, punto de partida de las contribuciones de esta tesis; por otra, la representación de la sintaxis y la semántica de los lenguajes de programación.

En el capítulo 3 se describe detalladamente la Evolución Gramatical en su forma original, y en el capítulo 4 se propone una serie de ajustes y modificaciones que ha sido necesario aplicar a dicha técnica antes de extenderla con la incorporación de aspectos semánticos.

Los capítulos 5 y 6 describen con detalle las contribuciones de la tesis. En el capítulo 5 se describe la primera extensión a Evolución Gramatical que consiste en la incorporación de semántica haciendo uso de gramáticas de atributos. En el capítulo 6, se describe la segunda extensión que hace uso de gramáticas de Christiansen para la incorporación de aspectos semánticos.

El capítulo 7 está dedicado a la aplicación **GEEMMA** , desarrollada para probar las técnicas propuestas en esta tesis.

Por último, en el capítulo 8 se muestran las conclusiones del trabajo, algunas aplicaciones del mismo, y propuestas para líneas futuras de investigación.

1.3. Publicaciones y referencias

Las publicaciones que se han realizado a partir del trabajo desarrollado en esta tesis son las siguientes.

Por una parte, la primera propuesta de esta tesis, extender con semántica la Evolución Gramatical con gramáticas de atributos está publicado en la siguiente referencia:

> M. de la Cruz, A. Ortega, y M. Alfonseca
> **Attribute grammar evolution**
> J. Mira and J.R. Alvarez, editors, Artificial Intelligence and Knowledge Engineering Applications:A Bioinspired Approach
> LNCS, volumen 3562, páginas 182-191. 2005. Springer

La segunda propuesta, extender con semántica la Evolución Grama-

tical con gramáticas de Christiansen está publicado en la siguiente referencia:

A. Ortega, M. de la Cruz y M. Alfonseca
Christiansen grammar evolution: Grammatical evolution with semantics
IEEE Transactions on Evolutionary Computation, 11(1), páginas 77-90. 2007.

Con el objetivo final de construir un sistema de programación automática de sistemas H, se ha publicado, en primer lugar, una gramática de Christiansen para definir dichos sistemas, y en segundo lugar, está admitida una comunicación en la que se describe un simulador para dichos sistemas. Las referencias de ambas publicaciones son las siguientes:

M. de la Cruz, A. Ortega
A Christiansen Grammar for Universal Splicing Systems
Third International Work-Conference on the Interplay Between
Natural and Artificial Computation. IWINAC 2009
LNCS, volumen 5601, páginas 336-345. 2009. Springer

J. M. Rojas Siles, M. de la Cruz y A. Ortega
Towards the automatic programming of H systems:
jHsys, a Java H system simulator
Admitido en PAAMS 2010

Una gramática de Christiansen para sistemas P, como primer paso para la programación automática de sistemas P se puede encontrar en la siguiente referencia:

A. Ortega, R. Núñez, M. de la Cruz y M. Alfonseca
Christiansen grammar for some P systems
Third Brainstorming Week on Membrane Computing
RGNC Report 01/2005 Research Group on Natural Computing
Universidad de Sevilla. Páginas 229-248. 2005. Fénix Editora

Se ha diseñado una gramática de Christiansen y se ha utilizado para el diseño automático de modelos matemáticos para el proceso de habituación en aprendizaje. Los resultados preliminares pueden encontrarse en:

del Rosal, E., Ortega, A., Alfonseca, M. y de la Cruz, M.
Christiansen Grammar Evolution for
the Modelling of Psychological Processes
Proccedings of the 5th International Industrial Simulation Conference
Páginas 99-104. 2007

Las publicaciones anteriores han dado lugar a 15 citas (excluyendo las autocitas).

Capítulo 2

Contexto

El objetivo de este capítulo es contextualizar el trabajo desarrollado en esta tesis. El núcleo fundamental de la tesis consiste en la extensión de una técnica de programación genética lineal denominada Evolución Gramatical. Esta técnica permite evolucionar programas en cualquier lenguaje de programación, y hace uso de gramáticas independientes del contexto para garantizar la clausura sintáctica. Las extensiones propuestas en esta tesis incorporan aspectos semánticos a dichas gramáticas para que los programas, además de ser correctos sintácticamente, lo sean también semánticamente. Es decir, podríamos decir que el objetivo es garantizar la clausura semántica. Por lo tanto, parece claro que la contextualización del trabajo se debe centrar en dos áreas diferentes: por una parte, en la descripción formal completa de los lenguajes de programación de alto nivel, y por otra, en la programación genética lineal. En cuanto a la descripción de los lenguajes de programación, se introduce la discusión tradicional relativa a la distinción entre sintaxis y semántica, y se presenta un conjunto de métodos para la expresión completa de los lenguajes de programación de alto nivel, es decir, de su sintaxis y su semántica. En concreto, se profundiza en los dos modelos elegidos en esta tesis para expresar la semántica: las gramáticas de atributos y las gramáticas de Christiansen. En lo referente a la programación genética lineal, se presenta una visión general del área donde se enmarca: la programación genética. Partimos de la propuesta original realizada por Koza, y presentamos una clasificación de los distintos sistemas de programación genética que se han propuesto en los últimos años. Dentro de dicha clasificación se presta especial atención a la categoría en la que se ubica la Evolución Gramatical. Por último, se describen algunos trabajos relacionados con la gestión de información semántica en el área de la programación genética.

2.1. Descripción formal de los lenguajes de programación de alto nivel

La descripción formal de los lenguajes de programación comparte aspectos comunes con la descripción del lenguaje natural, como por ejemplo los tres componentes que forman la definición de un lenguaje [Slonneger y Kurtz (1995)]:

- La **sintaxis** es la componente que define la manera en la que se combinan los símbolos del lenguaje para generar frases o programas bien formados. La sintaxis sólo está relacionada con la estructura de los símbolos del lenguaje, y no contempla de ninguna manera los aspectos relacionados con el significado.

- La **semántica** hace referencia al significado de las cadenas sintácticamente correctas.

- La **pragmática** contempla aspectos del lenguaje relacionados con el usuario del mismo, como por ejemplo, utilidad o ámbito de aplicación. En concreto, en los lenguajes de programación, la pragmática incluiría aspectos como la facilidad de implementación o la metodología de programación.

Dado que el objetivo de este trabajo es extender con semántica una técnica de programación genética que genera y evoluciona programas sintácticamente correctos, la pragmática queda fuera del alcance de este trabajo. En los siguientes apartados se presenta, en primer lugar, una discusión acerca de la distinción entre sintaxis y semántica de los lenguajes de programación. En segundo lugar se mencionan algunos de los métodos formales más extendidos para definir la sintaxis y la semántica de los lenguajes de programación.

2.1.1. Sintaxis y semántica de los lenguajes de programación

Desde que comenzó el estudio formal de los lenguajes de programación, existe la discusión acerca de lo que se denomina sintaxis y lo que es la semántica. Probablemente el origen de esta discusión es la búsqueda de métodos formales para definir los lenguajes de programación. Es evidente que si se quiere diseñar un método para definir formalmente la sintaxis de los lenguajes de programación, lo primero que hay que establecer es qué se entiende por sintaxis, y de la misma manera ocurre con la semántica.

Por una parte, parece que existe consenso en cuanto a la definición de la sintaxis de los lenguajes de programación de manera similar a la del lenguaje natural. De hecho, las gramáticas formales que propuso Chomsky para el lenguaje natural se aplican también a los lenguajes de programación [Chomsky (1956) y Chomsky (1959)]. Junto con la definición del concepto de gramática, Chomsky propuso también una clasificación de las mismas. Dentro de esta clasificación, las gramáticas independientes del contexto (o gramáticas de tipo 2) son las que se utilizan habitualmente para describir la sintaxis de los lenguajes de programación. La notación más extendida para la representación de dichas gramáticas es la notación BNF propuesta por John Backus y Peter Naur.

El siguiente ejemplo relativo a la dificultad de definir la linea que separa sintaxis de semántica se ha tomado prestado de [Slonneger y Kurtz (1995)]. Supóngase el siguiente programa:

```
program illegal is
    var a: boolean;
begin
     a:=5
end
```

La variable "a" ha sido declarada de tipo "boolean", y posteriormente aparece la asignación de un valor entero a la variable. El colectivo de desarrolladores de compiladores tiende a calificar este error como perteneciente a la *semántica estática* del lenguaje ya que se puede detectar a partir del texto del programa, es decir, es detectable en tiempo de compilación.

Los autores de [Slonneger y Kurtz (1995)] defienden que los errores estáticos pertenecen a la sintaxis del lenguaje y no a la semántica. Por ejemplo, si en un programa se realiza la siguiente declaración de una constante:

$$\textbf{const } c = 5;$$

Las siguientes asignaciones son erróneas esencialmente por el mismo motivo: no tiene sentido asignar un valor a una constante:

$$5 := 66;$$

$$c := 66;$$

Sin embargo, desde el punto de vista de un desarrollador de compiladores, el error de la primera asignación ($5 := 66$)se detectaría a partir de

la gramática independiente del contexto mientras que, el segundo, sería considerado como perteneciente a la semántica estática. En [Meek (1990)] se puede encontrar una discusión acerca del concepto de semántica estática.

Los autores de [Slonneger y Kurtz (1995)] proponen que se denomine sintaxis a todo aquello relativo al texto estático de los programas, y semántica a otros aspectos relacionados con el comportamiento del programa durante su ejecución. Con esta distinción entre sintaxis y semántica, se considerarían errores semánticos operaciones del tipo división por cero, acceso a variables no inicializadas, lectura de ficheros vacíos, bucles infinitos, etc. De esta manera, la sintaxis tiene dos componentes: la **sintaxis independiente del contexto** que se puede describir en BNF, y la **sintaxis dependiente del contexto** que hace referencia a las restricciones de contexto que tienen que cumplir cualquier programa válido.

Otros autores, como por ejemplo Marcotty [Marcotty y Bochmann (1976)], utilizan la misma definición de sintaxis independiente del contexto y sintaxis dependiente del contexto, reservando el término semántica para aspectos relacionados con el comportamiento del programa durante la ejecución del mismo. Sin embargo, es habitual encontrar en la literatura el uso indistinto de los términos semántica y sintaxis dependiente del contexto. En este trabajo también se hará así, denominaremos semántica a todo lo que no es independiente del contexto. Es decir, agruparemos en el término semántica lo que algunos autores denominan sintaxis dependiente del contexto y semántica. Esta elección relativa al convenio que se utiliza en este trabajo para separar sintaxis y semántica está motivada por el método de partida para las extensiones propuestas en esta tesis (Evolución Gramatical), ya que, dicho método utiliza una gramática independiente del contexto para describir la sintaxis de los programas, y por lo tanto, no tiene ningún sentido adoptar otra definición de sintaxis.

En los siguientes apartados se presentan algunos de los métodos más conocidos para definir de manera completa los lenguajes de programación, entendiendo por completa aquella definición que cubre tanto los aspectos sintácticos como los semánticos. La mayoría de ellos conserva como núcleo una gramática independiente del contexto, para no perder sus ventajas, que se extiende con componentes capaces de expresar la *semántica*. Precisamente esta característica es la que se ha buscado en los dos métodos elegidos en esta tesis para expresar la semántica, ya que el objetivo es añadir semántica a una técnica que utiliza una gramática independiente del contexto para asegurar la corrección sintáctica de los programas generados.

Los métodos se presentan clasificados en función de su adaptabilidad. Se entenderá que un método es adaptable si, como consecuencia de su propio uso, permite cambios en las estructuras que dan soporte al método. Por ejemplo, si un método utiliza una gramática independiente del contexto, y como consecuencia de la aplicación del método, dicha gramática puede variar, se considera que es adaptable.

Existen otros métodos de definición de la semántica que quedan fuera del alcance de esta tesis, como por ejemplo, la semántica operacional [Nielson y Nielson (1992)], la semántica denotacional [Gordon (1979)] y la semántica axiomática [Meyer (1990)].

2.1.2. Métodos estáticos o no adaptables

Los métodos estáticos o no adaptables que utilizan gramáticas se caracterizan por distinguir entre el diseño de la gramática y su uso. No se permite que la gramática se modifique mientras se utiliza en oposición a las especificaciones adaptables que se describen posteriormente. Los siguientes modelos son algunos de los más referenciados en la literatura:

- Gramáticas de van Wijngaarden
- Gramáticas de atributos
- *Definite Clause Grammars*

Se ha demostrado que las gramáticas de van Wijngaarden y las gramáticas de atributos son equivalentes [Dembinski y Maluszynski (1978)], y que también lo son las gramáticas de atributos y las *definite clause grammars* [Deransart y Maluszynski (1985)]. Por ese motivo, realmente las diferencias entre ellas son más bien de notación y enfoque.

2.1.2.1. Gramáticas de van Wijngaarden

Las gramáticas de van Wijngaarden también son conocidas como gramáticas W o gramáticas de dos niveles. Fueron desarrolladas por Adriaan van Wijngaarden en los años 60 para la descripción formal completa del lenguaje de programación ALGOL68 [van Wijngaarden (1977)]. El objetivo del autor era expresar mediante una gramática de dos niveles la sintaxis y la semántica de ALGOL68.

La descripción original del formalismo se puede consular en [van Wijngaarden (1965)], pero, para mayor claridad, se recomienda consultar [Marcotty y Bochmann (1976)], una revisión sobre distintas aproximaciones para la descripción de lenguajes de programación.

Una gramática W consiste en dos conjuntos de reglas, las metaproducciones y las hiperreglas. La combinación permite la generación de un conjunto de producciones potencialmente infinito que define la sintaxis y las restricciones dependientes del contexto.

2.1.2.2. Gramáticas de atributos

Las gramáticas de atributos se describen con mayor amplitud en la sección 2.1.4. Es el formalismo elegido en este trabajo para expresar restricciones semánticas en el primero de los dos métodos propuestos en esta tesis.

2.1.2.3. *Definite Clause Grammars*

Este modelo constituye una interfaz adecuada para utilizar gramáticas de atributos en el lenguaje de programación lógica Prolog. La transformación de una gramática expresada con este formalismo a un programa Prolog es un proceso mecánico. En la actualidad la mayoría de los intérpretes de Prolog incorporan DCGs y cualquier manual de Prolog o programación lógica [Bratko (1990), Nilsson y Maluszynski (1990), Pereira y Warren (1980)] describe completamente todas sus características.

2.1.3. Métodos adaptables

Los intentos de definición de gramáticas independientes del contexto con capacidad de modificarse, comienzan a la vez que el diseño de lenguajes de programación y la automatización de la construcción de compiladores e intérpretes. Según la revisión que se puede consultar en [Christiansen (1990)] podemos considerar que Alfonso Caracciolo di Forino, en 1963 [di˜Forino (1963)], fue el primero en mencionar esta posibilidad.

En la actualidad se pueden clasificar las descripciones adaptables de la siguiente manera:

- **Imperativas**: No son el objetivo de este trabajo por lo que sólo se enumerará brevemente las características generales de las propuestas más referenciadas. Pueden encontrarse más detalles en algunos trabajos que incluyen revisiones más amplias [Shutt (1993) y Christiansen (1990)]. Sus características más relevantes son las siguientes:
  - Dependen del algoritmo concreto de análisis que se utilice. Todas suponen que su uso va a ser para análisis sintáctico.
  - Parece que el objetivo de todos estos modelos es asegurar al analizador sintáctico la próxima decisión que debe tomar. De esta

forma, el diseñador de la gramática está obligado a conocer perfectamente el orden en el que el analizador va a actuar.

- Se mencionará brevemente las siguientes propuestas:
  - Las gramáticas independientes del contexto y extensibles (Extensible Context Free Grammars) de Wegbreit.
  - Los "Dynamic Template Translators" de Mason.
  - Las gramáticas modificables de Burshteyn.

- **Declarativas**: Estas propuestas no suponen ningún algoritmo de análisis concreto. Podemos mencionar las siguientes aproximaciones:
  - Gramáticas de Christiansen.
  - Gramáticas adaptables (o adaptativas) recursivas de Shutt.

Aunque también describiremos muy brevemente alguna características de la propuesta de Shutt, nuestro trabajo sólo utilizará, de este grupo, el modelo de Christiansen.

Se mencionó anteriormente las demostraciones de equivalencia [Dembinski y Maluszynski (1978) y Deransart y Maluszynski (1985)] de las gramáticas de atributos, las de van Wijngaarden y las DCGs. Dadas estas equivalencias, Christiansen sugiere en [Christiansen (1990)] la posibilidad de extender cualquier otro modelo con la misma filosofía que sus gramáticas. Posteriormente, en [Christiansen (2009)] él mismo extiende de este modo las DCGs en Prolog.

2.1.3.1. Las ECFGs de Wegbreit

Las ECFGs de Wegbreit [Wegbreit (1970)] añaden una máquina de estado finito a las gramáticas independientes del contexto para gobernar el proceso de análisis sintáctico. Wegbreit estaba interesado en la definición de un lenguaje de programación extensible (de nombre EL1). Las ECFGs generan una familia de lenguajes que se sitúa propiamente entre los independientes del contexto y los recursivamente enumerables. Por lo tanto, no tienen la potencia expresiva de las gramáticas de tipo 0 de Chomsky.

2.1.3.2. Los DTTs de Mason

En este modelo [Mason (1984) y Mason (1987)] cada regla tiene indicaciones de qué otras reglas añadir o quitar de la gramática. Se añade un nivel intermedio ("templates") que dificulta el proceso de derivación y oscurece el diseño de las gramáticas de una forma similar a otras propuestas que añaden niveles intermedios a la derivación, como las gramáticas de van Wijngaarden.

2.1.3.3. Las gramáticas modificables de Burshteyn

Se pueden describir como una modificación de las ECFGs en las que se utiliza una máquina de Turing en lugar de una máquina de estado finito. Se supera así la limitación de la potencia expresiva mencionada anteriormente.

2.1.3.4. Gramáticas adaptativas recursivas de Shutt

En la revisión de las gramáticas adaptables que Shutt incluye en su tesis doctoral [Shutt (1993)], él mismo considera que las gramáticas de Christiansen son la única propuesta de gramática adaptable que puede ser considerada como tal. También escribe literalmente en "Motivation for the thesis":

> "the purpose of this thesis is to develop a Turing-powerfull adaptable grammar model that preserves, as much as possible, the elegance and simplicity of CFGs."

La razón por la que Shutt considera que los enfoques basados en gramáticas de atributos, como el de Christiansen, pierden gran parte de la elegancia y simplicidad de las gramáticas independientes del contexto, se debe a que las acciones semánticas que calculan los valores de los atributos permiten la realización de cualquier tarea, ya que cualquier función y lenguaje de programación puede ser utilizado para describirlas. Shutt se plantea como objetivo reducir todo el proceso (tanto sintáctico como semántico) a la operación de derivación. De esta manera no se pierde la elegancia y simplicidad que menciona.

2.1.4. Gramáticas de atributos

Las gramáticas de atributos fueron propuestas por Donald Knuth a finales de los años 60 [Knuth (1968) y Knuth (1971)] como formalismo para la representación de restricciones dependientes del contexto de los lenguajes de programación. En la extensa literatura perteneciente al área de los lenguajes formales, se pueden encontrar múltiples y distintas definiciones y notaciones del formalismo. Algunas de ellas mantienen la idea original mientras que otras incorporan ampliaciones. En el contexto de esta tesis se utilizará la definición y la notación propuesta en [Alfonseca y otros (2007)] que a continuación se reproduce literalmente.

Se llama gramática de atributos a una extensión de las gramáticas independientes del contexto a las que se añade un sistema de atributos. Un sistema de atributos está formado por:

- *Un conjunto de atributos semánticos que se asocia a cada símbolo de la gramática.*
- *Los datos globales de la gramática, accesibles desde cualquiera de sus reglas, pero no asociados a ningún símbolo concreto.*
- *Un conjunto de acciones semánticas, distribuidas por las reglas de producción.*

De forma semejante a las variables en los lenguajes de programación de alto nivel, un atributo semántico se define como un par, compuesto por un tipo de datos (la especificación de un dominio o conjunto) y un nombre o identificador. En algunos ejemplos de este capítulo, debido a su simplicidad, el dominio de los atributos es irrelevante y se omitirá. En cada momento, cada atributo semántico puede tener un valor único que tiene que pertenecer a su dominio. Dicho valor puede modificarse en las acciones semánticas, sin ningún tipo de restricción.

Una acción semántica es un algoritmo asociado a una regla de la gramática de atributos, cuyas instrucciones sólo pueden referirse a los atributos semánticos de los símbolos de la regla y a la información global de la gramática de atributos. El objetivo de la acción semántica es calcular el valor de alguno de los atributos de los símbolos de su regla, sin restricciones adicionales.

En este capítulo se utilizará la siguiente notación para las gramáticas de atributos:

$$A = \{\Sigma_T, \Sigma_N, S, P, K\}$$

- *Se mantiene la estructura de las gramáticas independientes del contexto.*
- *A cada símbolo de la gramática le acompaña la lista de sus atributos semánticos entre paréntesis. El nombre de cada atributo sigue al de su dominio [...] En los símbolos sin atributos semánticos se omitirán los paréntesis.*
- *Las reglas de producción se modifican para distinguir distintas apariciones del mismo símbolo. A cada regla de producción le acompaña su acción semántica [...] Las instrucciones de la acción se escriben entre*

llaves. Para referirse a un atributo semántico de un símbolo dentro de las acciones semánticas, se escribirá el nombre del atributo tras el del símbolo, separados por un punto. Si alguna regla no necesita realizar ninguna acción semántica, se escribirá "{}". La información global se añade como nueva componente adicional, al final de la gramática de atributos (K).

Los atributos semánticos se pueden clasificar según la posición, en la regla de producción, de los símbolos cuyos atributos se utilicen en el cálculo. Se distinguen dos grupos:

- ***Atributos sintetizados:*** *son los atributos asociados a los símbolos no terminales de la parte izquierda de las reglas de producción cuyo valor se calcula utilizando los atributos de los símbolos que están en la parte derecha correspondiente.*
- ***Atributos heredados:*** *El resto de las situaciones origina atributos heredados, es decir, atributos asociados a símbolos de la parte derecha de las reglas cuyo valor se calcula utilizando los atributos de la parte izquierda o los de otros símbolos de la parte derecha. En este grupo es necesaria la siguiente distinción:*
  - *Atributos heredados por la derecha, cuando el cálculo del valor de un atributo utiliza atributos de los símbolos que están situados a su derecha.*
  - *Atributos heredados por la izquierda, cuando sólo se utilizan los que están a su izquierda, ya sea en la parte derecha de la regla o en la parte izquierda.*

Como ilustración del uso de las gramáticas de atributos para expresar restricciones dependientes del contexto, utilizaremos un ejemplo que aparece en [Shutt (1993)]. En dicho ejemplo se describe un lenguaje de programación cuya sintaxis está definida por la gramática independiente del contexto $G_I = \{\Sigma_T, \Sigma_N, S, P\}$, donde el conjunto de símbolos terminales Σ_T es:

$$\Sigma_T = \{int, ;, =, \{, \}, a, b, .., z\}$$

el conjunto Σ_N de no terminales es,

$$\Sigma_N = \{ \quad programa, declaraciones, declaracion, identificador, letra, \\ sentencias, sentencia \}$$

y el axioma,

$$S = programa$$

Por último, el conjunto de reglas P contiene los siguientes elementos:

$$<programa> \rightarrow \{<declaraciones><sentencias>\}$$
$$<declaraciones> \rightarrow \lambda$$
$$<declaraciones> \rightarrow <declaracion><declaraciones>$$
$$<declaracion> \rightarrow int <identificador>;$$
$$<sentencias> \rightarrow \lambda$$
$$<sentencias> \rightarrow <sentencia><sentencias>$$
$$<sentencia> \rightarrow <identificador>=<identificador>;$$
$$<identificador> \rightarrow <letra>$$
$$<identificador> \rightarrow <letra><identificador>$$
$$<letra> \rightarrow a \mid b \mid \cdots \mid z$$

La especificación del lenguaje exige que los identificadores sólo se puedan declarar una vez en cada programa y que no se puedan utilizar si no han sido previamente declarados. Estas dos condiciones son dependientes del contexto. A continuación se muestran unos ejemplos de programas escritos en el lenguaje ejemplo.

$\{$ *int x; int y;* $x = y;$ $\}$
$\{$ *int x; int x;* $\}$
$\{$ *int x; int z;* $x = y;$ $\}$

Los tres programas son sintácticamente correctos, pero dos de ellos no cumplen las restricciones dependientes del contexto. El segundo no es válido porque el identificador x se declara dos veces. El tercero es incorrecto porque el identificador y no está declarado y es usado.

A continuación se describe una gramática de atributos que permite expresar las restricciones dependientes del contexto del lenguaje. La idea básica es construir, utilizando el sistema de atributos, una lista de identificadores definidos, y utilizar esta lista para comprobar la unicidad de las declaraciones, y el uso de identificadores que hayan sido previamente declarados.

Los atributos asociados a los símbolos son los siguientes:

- El símbolo no terminal <*identificador*> tiene un atributo sintetizado llamado *cadena* para guardar el lexema (o nombre) del identificador.
- El símbolo no terminal <*letra*> tiene un atributo llamado *valor* para guardar su valor (la propia letra).
- Los símbolos <*declaracion*> y <*declaraciones*> tienen dos atributos. El primero, *prelista*, es heredado y contiene una lista con todos los indentificadores declarados antes de la derivación del no terminal. El segundo, *postlista*, es sintetizado y contiene la lista de los identificadores declarados antes y después de la derivación del no terminal.
- Los no terminales <*sentencia*> y <*sentencias*> tienen un atributo llamado *lista* que contiene la lista de todos los identificadores declarados en la sección declarativa del programa.

Respecto a las acciones semánticas para la evaluación de los atributos, en primer lugar, el símbolo <*identificador*> sintetiza en su atributo *cadena* su lexema. Las acciones relacionadas con dicha síntesis son las siguientes:

<*identificador*> → <*letra*>
  {
    <*identificador*> .*cadena* =<*letra*> .*valor*
  }
<*identificador*>$_1$ → <*letra*><*identificador*>$_2$
  {
    <*identificador*>$_1$.*cadena* =<*letra*> .*valor* ||[1] <*identificador*>$_2$.*cadena*
  }
<*letra*> → *a*
  {
    <*letra*> .*valor* = " *a* "
  }
.....
<*letra*> → *z*
  {
    <*letra*> .*valor* = " *z* "
  }

En segundo lugar, las acciones semánticas relacionadas con la restricción de la unicidad de los identificadores implica, por una parte, inicializar a vacío

[1] El símbolo || representa concatenación de cadenas de caracteres.

la lista de identificadores declarados en la regla del axioma; por otra parte, cada vez que se declara un identificador, se añade a la lista siempre que no haya sido previamente declarado. Las acciones semánticas son las siguientes:

<*programa*> → {<*declaraciones*><*sentencias*>}
{
<*declaraciones*> .*prelista* = Φ
}
<*declaracion*> → *int* <*identificador*>;
{
SI (<*identificador*> .*cadena* ∈<*declaracion*> .*prelista*)
INCUMPLIMIENTO RESTRICCIÓN UNICIDAD
SI NO
<*declaracion*> *postlista* =<*identificador*> .*cadena* ∪ <*declaracion*> .*prelista*
}
<*declaraciones*>$_1$ → <*declaracion*><*declaraciones*>$_2$
{
<*declaracion*> .*prelista* =<*declaraciones*>$_1$.*prelista*
<*declaraciones*>$_2$.*prelista* =<*declaracion*> .*postlista*
<*declaraciones*>$_1$.*postlista* =<*declaraciones*>$_2$.*postlista*
}
<*declaraciones*> → λ
{
<*declaraciones*> .*postlista* =<*declaraciones*> .*prelista*
}

Por último, las acciones semánticas relacionadas con la restricción relativa a la declaración de identificadores antes de su uso son las siguientes. Sólo se requiere la propagación de la lista de identificadores declarados desde el bloque de declaraciones del programa hasta el bloque de sentencias. Las acciones son:

<*programa*> → {<*declaraciones*><*sentencias*>}
{
<*sentencias*> .*lista* =<*declaraciones*> .*postlista*
}
<*sentencias*>$_1$ → <*sentencia*><*sentencias*>$_2$
{
<*sentencia*> .*lista* =<*sentencias*>$_1$.*lista*
<*sentencias*>$_2$.*lista* =<*sentencias*>$_1$.*lista*
}
<*sentencia*> → <*identificador*>$_1$=<*identificador*>$_2$
{
SI (<*identificador*>$_1$.*cadena* ∉<*sentencia*> .*lista*) OR

$(<identificador>_2 .cadena \notin <sentencia> .lista)$
INCUMPLIMIENTO RESTRICCIÓN DECLARACIÓN
}
$<sentencias> \rightarrow \lambda \{\}$

Una vez definidos los atributos y sus funciones de evaluación en las acciones semánticas, es necesario establecer el orden de evaluación de los mismos, ya que, para evaluar correctamente un atributo, resulta imprescindible la evaluación previa de los atributos de los que depende. Los grafos de dependencias son una herramienta adecuada para determinar el orden de evaluación. Se puede consultar el algoritmo de construcción en [Aho y otros (2008) y Grune y otros (2007)]. Otra técnica para garantizar el orden de evaluación adecuado es hacer uso de unos subconjuntos de gramáticas de atributos que aseguran órdenes de evaluación con resultados correctos, las gramáticas S-atribuidas y las gramáticas L-atribuidas. En [Aho y otros (2008) y Grune y otros (2007)] se puede consultar la definición de las mismas. Brevemente, las gramáticas S-atribuidas son aquellas en las que todos los atributos son sintetizados. La evaluación de una gramática S-atribuida se puede hacer con un recorrido en postorden de los árboles sintácticos atribuidos. Las gramáticas L-atribuidas son aquellas en las que los atributos son sintetizados y heredados con alguna restricción. En cada producción, los atributos heredados por un hijo sólo dependan de los atributos heredados de su padre, y de los heredados/sintetizados de los hermanos situados a su izquierda. De esta manera, se puede evaluar correctamente una gramática L-atribuida con un recorrido en preorden para evaluar los atributos heredados de padres a hijos, y un recorrido en postorden para evaluar los atributos sintetizados y los heredados entre hermanos.

En los dos métodos propuestos en esta tesis se utiliza la segunda aproximación para asegurar la evaluación correcta de los atributos. En concreto se propone un recorrido en profundidad por la izquierda con retroseguimiento que permite evaluar correctamente gramáticas L-atribuidas. De hecho, nuestro algoritmo de recorrido permite evaluar un tipo más general de gramáticas. El retroseguimiento permite que un símbolo también herede información que su padre ha sintetizado de sus hermanos izquierdos.

2.1.5. Gramáticas de Christiansen

Este formalismo fue propuesto por Henning Christiansen a mediados de los años 80 como parte de su tesis doctoral [Christiansen (1985), Christiansen (1986), Christiansen (1988b) y Christiansen (1988a)]. Inicialmente las llamó "gramáticas generativas", pero otros autores las referenciaron como

"gramáticas adaptativas" y "gramáticas de Christiansen", siendo la última denominación la que se ha consolidado como definitiva.

El autor define el formalismo en [Christiansen (1988b)] como una extensión de las gramáticas de atributos en la que el sistema de atributos es responsable de la generación y gestión de nuevas reglas sintácticas y la relación de derivación se sustituye por otra más potente y sensible al contexto.

La diferencia entre las gramáticas de atributos y las gramáticas de Christiansen, según se describe en [Christiansen (1988b)] es la siguiente. En una gramática de atributos, la derivación, es decir, la sustitución de un no terminal por una cadena de símbolos terminales y no terminales, se realiza de acuerdo al conjunto fijo de reglas de la gramática. El proceso de derivación consiste en buscar en ese conjunto fijo una regla para el no terminal, y sustituirlo por la parte derecha de dicha regla. En una gramática de Christiansen, por convenio, el primer atributo de todos los símbolos no terminales es heredado y contiene una gramática completa. Dicha gramática es la que se utiliza en la derivación. La derivación de una cadena completa comienza con la asignación de la gramática inicial al primer atributo del axioma.

La definición formal de las gramáticas de Christiansen se pueden consultar en las referencias previamente citadas. En cuanto a la notación del formalismo, en este trabajo se ha optado por mantener la notación de gramáticas de atributos descrita en la sección 2.1.4 en lugar de utilizar la propuesta original. En dicha propuesta, en las reglas, los símbolos no terminales van acompañados de sus correspondientes atributos encerrados entre paréntesis. Además, los atributos sintetizados van precedidos del símbolo $\uparrow$, y los heredados del símbolo $\downarrow$. Por último, los nombres de los atributos se utilizan de manera similar a las variables en programación lógica (por ejemplo Prolog). En efecto, la coincidencia de los nombres de las variables que ocupan la posición de los atributos en diferentes lugares de las acciones semánticas expresan asignaciones entre ellos.

El ejemplo que se presentó como ilustración del uso de gramáticas de atributos para la expresión de restricciones dependientes del contexto, se describe a continuación utilizando el formalismo de Christiansen. La restricción relativa a la definición previa de un identificador antes de su uso se puede expresar fácilmente con este formalismo. Sin embargo, la restricción de la unicidad de los identificadores es más complicada de formalizar. Aunque más adelante se esboza una alternativa que permite expresar la segunda restricción, por razones de simplicidad, se modifica el lenguaje limitando a uno la longitud de los identificadores. Atendiendo a esta simplificación, las reglas de la gramática

independiente del contexto serían las siguientes:

$$
\begin{aligned}
&<programa> \rightarrow \{<declaraciones><sentencias>\}\\
&<declaraciones> \rightarrow \lambda\\
&<declaraciones> \rightarrow <declaracion><declaraciones>\\
&<declaracion> \rightarrow int\ <identificador>;\\
&<sentencias> \rightarrow \lambda\\
&<sentencias> \rightarrow <sentencia><sentencias>\\
&<sentencia> \rightarrow <identificador>=<identificador>;\\
&<identificador> \rightarrow <letra>\\
&<letra> \rightarrow a \mid b \mid \cdots \mid z
\end{aligned}
$$

A continuación se describe cómo expresar con gramáticas de Christiansen las restricciones dependientes del contexto del ejemplo que nos ocupa. Es importante recordar que en este formalismo la gramática se adapta para generar cadenas que cumplan las restricciones establecidas. La adaptación se produce como consecuencia del uso de la propia gramática en el proceso de derivación de una cadena del lenguaje.

En primer lugar, para asegurar que los identificadores se declaran antes de su uso, se modificará la gramática para que durante todo el proceso de derivación de una cadena, sólo estén disponibles los identificadores declarados. Obsérvese que la regla de $<declaracion>$ es la que introduce el no terminal $<identificador>$. Como es necesario poder declarar cualquier letra como identificador, se sustituirá el no terminal $<identificador>$ por $<letra>$ y se añadirán las reglas para $<identificador>$ en el momento en el que se sepa el identificador concreto que se ha declarado, es decir, como una acción semántica asociada a la regla de $<declaracion>$. Por lo tanto es necesario eliminar en la gramática inicial la regla que define un identificador como una letra. Con esta modificación, la gramática quedaría como sigue:

$$
\begin{aligned}
&<programa> \rightarrow \{<declaraciones><sentencias>\}\\
&<declaraciones> \rightarrow \lambda\\
&<declaraciones> \rightarrow <declaracion><declaraciones>\\
&<declaracion> \rightarrow int\ <letra>;\\
&<sentencias> \rightarrow \lambda\\
&<sentencias> \rightarrow <sentencia><sentencias>\\
&<sentencia> \rightarrow <identificador>=<identificador>;\\
&<letra> \rightarrow a \mid b \mid \cdots \mid z
\end{aligned}
$$

Obsérvese que la gramática "queda incompleta" en cuanto que inicialmente no dispone de producciones para el no terminal $<identificador>$. Se

completará como consecuencia del proceso de adaptación. La corrección del conjunto se asegura de la siguiente manera: como cada no terminal tendrá las reglas que puede aplicar, será suficiente con la gramática anterior durante la primera mitad del programa (la mitad derivada de $<declaraciones>$). Tras terminar esta parte, se habrán añadido las reglas necesarias para $<identificador>$ por lo que se tendrán disponibles cuando se llegue a la primera aparición de $<identificador>$ en una sentencia de asignación. Todo este proceso se realiza de manera análoga al tratamiento de cualquier atributo semántico en una gramática de atributos. Sólo es necesario añadir las operaciones de manipulación de la gramática. Supondremos las siguientes funciones que realizan la operación que sugiere su nombre:

GC añadirRegla(GC g, REGLA r)
GC eliminarRegla(GC g, REGLA r)

Ambas funciones tiene como parámetros de entrada una gramática de Christiansen, g, y una regla, r. La salida de las funciones es también una gramática de Christiansen que se obtiene ejecutando la acción correspondiente (añadir o eliminar).

A continuación se describe, informalmente, la gramática de Christiansen.

En primer lugar, llamaremos gh al primer atributo de todos los no terminales cuyo valor es la gramática heredada que contiene las reglas para derivar el no terminal. En la notación original este atributo se llamaría $\downarrow gh$.

El axioma de la gramática, <*programa*> sólo tiene un atributo, gh, que es precisamente la gramática de Christiansen inicial. En esta gramática inicial no existe ninguna regla para el no terminal $<identificador>$.

Los identificadores declarados en la primera parte de un programa, es decir, en el bloque correspondiente al no terminal <*declaraciones*> deben estar disponibles para su uso en la segunda, es decir, en el bloque correspondiente al no terminal <*sentencias*>. Por lo tanto, el no terminal <*declaraciones*> además de la gramática heredada de su primer atributo, gh, tiene asociado un segundo atributo sintetizado, gs, que también es una gramática de Christiansen. Esta segunda gramática sintetizada se construye añadiendo a la heredada, gh, las reglas asociadas a $<identificador>$. Por su parte, el no terminal <*sentencias*> recibe en su primer atributo la gramática sintetizada por <*declaraciones*>. Expresado en términos de acciones semánticas sería:

<*programa*> → {<*declaraciones*><*sentencias*>}
{
<*declaraciones*> .gh =<*programa*> .gh
<*sentencias*> .gh =<*declaraciones*> .gs

}

La regla anterior expresada con la notación original sería:

$$<programa(\downarrow gh)> \rightarrow \{<declaraciones(\downarrow gh, \uparrow gs)><sentencias(\downarrow gs)>\}$$

Puede observarse que el uso de *gs* en *sentencias* en la posición de su atributo *gh* indica la asignación $sentencias.gh = declaraciones.gs$. Este es el estilo declarativo que las unificaciones lógicas añaden a la notación original de las gramáticas de Christiansen. La propagación a todas las sentencias de los identificadores disponibles se hace fácilmente mediante herencia según se muestra en las siguientes acciones semánticas:

$<sentencias> \rightarrow \lambda \{\}$
$<sentencias>_1 \rightarrow <sentencia><sentencias>_2$
{
$<sentencia>.gh = <sentencias>_1.gh$
$<sentencias>_2.gh = <sentencias>_1.gh$
}
$<sentencia> \rightarrow <identificador>_1 = <identificador>_2;$
{
$<identificador>_1.gh = <sentencia>_1.gh$
$<identificador>_2.gh = <sentencia>_1.gh$
}

Por último, sólo falta describir la gestión de la declaración de los identificadores. El objetivo es que queden expresadas las dos restricciones dependientes del contexto. Por una parte, para asegurar que sólo se usan los identificadores declarados, basta con añadir a la gramática la regla correspondiente del no terminal $<identificador>$ cada vez que un identificador sea declarado. Por otra parte, para preservar la unicidad de los identificadores, cada vez que se declara uno de ellos, se elimina la correspondiente regla del no terminal *<letra>*. La declaración de un identificador queda reflejada en la regla del no terminal *<declaracion>*, por lo tanto, la acción semántica de dicha regla es el punto adecuado para añadir y eliminar las reglas que se han descrito. Además, es necesario propagar las modificaciones generadas en cada declaración para que sea correcta la información que recibe el no terminal *<sentencias>* acerca de los identificadores declarados. Todas estas ideas se expresan en las acciones semánticas de la siguiente manera:

$<declaraciones>_1 \rightarrow <declaracion><declaraciones>_2$
{
$<declaracion>.gh = <declaraciones>_1.gh$

$<declaraciones>_2$ .gh =<declaracion> .gs
$<declaraciones>_1$ .gs =$<declaraciones>_2$ .gs
}
<declaraciones> → λ
{
<declaraciones> .gs =<declaraciones> .gh
}
<declaracion> → int <letra>;
{
<letra> .gh =<declaracion> .gh
<declaracion> .gs = añadirRegla(<declaracion> .gh,
<identificador> → <letra> .valor)
<declaracion> .gs = eliminarRegla(<declaracion> .gs,
<letra> → <letra> .valor)
}
<letra> → a
{
<letra> .valor = " a "
}
.....
<letra> → z
{
<letra> .valor = " z "
}

En la figura 2.1 se muestra el árbol de derivación del siguiente programa:

$$\{\ int\ a;\ int\ b;\ a = b;\ \}$$

En color azul se han representado las dependencias entre atributos heredados, y en rojo las correspondientes a los sintetizados.

Dado que en las gramáticas de Christiansen cada no terminal tiene un primer atributo que contiene las reglas disponibles para su derivación, y este aspecto es fundamental ya que es la expresión de la adaptación de la gramática, a continuación se describen los cambios en la gramática producidos a lo largo del árbol de la figura 2.1. Para simplificar la exposición, no se describirá la gramática completa de cada no terminal. En su lugar, sólo se resaltarán los aspectos relacionados con las reglas que pueden variar, que son las reglas del no terminal <*letra*> y las del no terminal <*identificador*>.

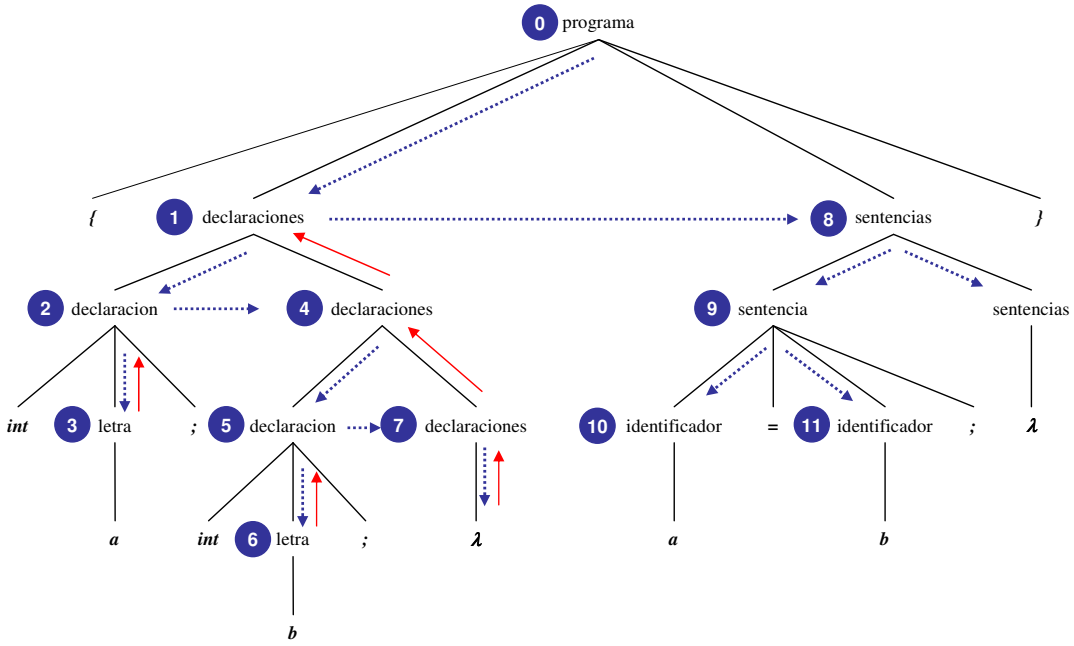

Figura 2.1: Árbol de derivación de la cadena { int a; int b; a = b; }.

En el punto 0, la gramática almacenada en el primer atributo del axioma, <*programa*>, no contiene ninguna regla para el símbolo <*identificador*>. Sin embargo, contiene todas las reglas para el no terminal <*letra*>, es decir, las reglas <*letra*> $\rightarrow$ *a* | .. | *z*.

En los puntos 1, 2 y 3 los no terminales <*declaraciones*>, <*declaracion*> y <*letra*> respectivamente, contiene en su primer atributo la misma gramática que el axioma ya que la han heredado de él, a través de una cadena de herencia.

En el punto 4, la gramática almacenada en el primer atributo es ya una versión adaptada de la gramática original del axioma. Efectivamente, en este punto, está disponible el identificador "*a*" y ya no es posible definirlo de nuevo. La adaptación se ha producido como consecuencia del uso de la regla <*letra*> $\rightarrow$ *a* y de la regla de <*declaracion*>. Dicho uso ha desencadenado la incorporación de la regla <*identificador*> $\rightarrow$ *a* lo que implica la disponibilidad del identificador "*a*", y la eliminación de la regla <*letra*> $\rightarrow$ *a*, lo imposibilita la redefinición del identificador "*a*".

En los puntos 5 y 6 los no terminales <*declaracion*> y <*letra*> contienen

en su primer atributo la misma gramática descrita en el punto 4 ya que la han heredado. En el punto 6, es imposible que se derive la letra "a" y por lo tanto también es imposible que se derive un identificador con ese lexema. Así se consigue satisfacer la restricción relativa a la unicidad de los identificadores.

En el punto 7, la gramática almacenada en el primer atributo del no terminal $<declaraciones>$ es una nueva versión adaptada. Contiene las reglas $<identificador> \rightarrow \ a \mid b$ y las reglas $<letra> \rightarrow \ c \mid .. \mid z$. Es decir, están disponibles los identificadores "a" y "b", y además, no puede redefinirse ninguno de ellos.

En los puntos 8, 9, 10 y 11, la gramática almacenada en el primer atributo del no terminal correspondiente contiene las reglas $<identificador> \rightarrow \ a \mid b$ y las reglas $<letra> \rightarrow \ c \mid .. \mid z$. Por lo tanto, en los puntos 10 y 11 sólo se pueden derivar los identificadores "a" y "b", lo que hace imposible violar la restricción relativa a la declaración de los identificadores antes de su uso.

Para terminar la exposición del ejemplo hay que mencionar la posible solución al caso en el que los identificadores no tengan limitada su longitud a una letra. En el caso general, en un instante determinado, se dispondrá de una gramática independiente del contexto para los identificadores disponibles. Tras declarar correctamente un nuevo identificador, bastará con obtener la gramática que genera el mismo conjunto de identificadores excepto el recién declarado. En general, se usan patrones regulares para definir los identificadores. Las propiedades de cierre de los lenguajes regulares aseguran que todos los conjuntos de identificadores válidos durante el proceso son regulares, por lo que siempre será posible especificar la gramática independiente del contexto que genera el conjunto de identificadores disponibles en cada momento.

2.2. Programación genética

La programación genética propuesta por Koza en la década de los 90 pertenece a la familia de los algoritmos evolutivos. Es un algoritmo orientado a la programación automática en el lenguaje de programación LISP. En los siguientes apartados se describe brevemente la propuesta de Koza y se presenta una clasificación de las múltiples variantes de dicha propuesta que han aparecido en los últimos años.

2.2.1. La programación genética propuesta por Koza

En esta sección se describe la propuesta de Koza de manera breve, ya que queda fuera del alcance de esta tesis profundizar en la exposición de esta

importante propuesta, que por otra parte, está extensamente documentada. La programación genética de Koza [Koza (1992), Koza (1994), Koza (1999) y Koza (2005)] es un algoritmo evolutivo que permite generar automáticamente programas escritos en LISP.

Según se describe en [Eiben y Smith (2003)], un algoritmo evolutivo se basa en la siguiente idea: dada una población de individuos, la presión del entorno origina el proceso de selección natural (supervivencia del mejor adaptado) lo que provoca un incremento en la aptitud/adaptación de la población. Inicialmente se crea de manera aleatoria una población de soluciones candidatas al problema que se quiere resolver. Se asigna a cada una de ellas un valor que representa su aptitud como solución del problema, se puede considerar que cuanto más alto es ese valor, mejor es la solución. Al valor de aptitud se le denominará *fitness*. Se seleccionan los mejores candidatos, es decir, los que tengan mejor *fitness* para crear la siguiente generación aplicándoles operadores genéticos, como la recombinación y/o la mutación. La recombinación es un operador que se aplica a dos o más candidatos (llamados padres) para generar uno o más candidatos (los hijos). La mutación se aplica a un candidato para obtener otro. La aplicación de la recombinación y la mutación, da lugar a un conjunto de nuevos candidatos (la progenie) que compite con los antiguos para formar parte de la nueva generación. Este proceso se itera hasta que se encuentra un candidato que tenga la suficiente calidad, o hasta que se alcance algún límite computacional previamente establecido.

El primer paso en el diseño de un algoritmo evolutivo es ligar el contexto del problema original con el espacio donde verdaderamente se desarrolla el proceso evolutivo. Los objetos que pertenecen al conjunto de posibles soluciones en el contexto del problema se denominan *fenotipos*, y su codificación correspondiente en el espacio evolutivo, es decir, los individuos de la población, se denominan *genotipos*. El espacio de fenotipos puede ser muy diferente del espacio de genotipos, pero es en éste último es donde se realiza la búsqueda evolutiva. Existen muchos sinónimos para nombrar a los elementos de ambos espacios. Cada elemento del espacio de fenotipos se puede denominar "solución candidata", "fenotipo" o "individuo". Los elementos del espacio de genotipos, se pueden referir como "genotipo", "cromosoma" o también "individuo". Por otra parte, es habitual denominar "representación" al enlace que se establece entre los dos espacios, de manera que un genotipo *representa* a un fenotipo, es decir, la palabra "representación" sería sinónimo de "codificación". Sin embargo, también se utiliza el término representación para hacer referencia a la estructura de datos del genotipo. Atendiendo a esta última acepción, los algoritmos evolutivos se pueden clasificar de la siguiente manera. En los *algoritmos genéticos*, la representación consiste en una cadena de símbolos de un alfabeto finito. En las *estrategias evolutivas*, los individuos se

representan con vectores de números reales. La *programación evolutiva* utiliza como representación los autómatas finitos. Y por último, *la programación genética* hace uso de árboles para representar la estructura de los programas que forman la población.

Obviamente los operadores de recombinación y mutación deben adaptarse a la representación de los individuos.

La propuesta concreta de Koza en cuanto a la representación de los individuos y los operadores de recombinación y mutación se resume a continuación.

Los individuos son expresiones simbólicas (*s-expressions*) representadas en forma de árbol. Cada nodo del árbol es una *función* o un *terminal*. Las funciones realizan operaciones sobre terminales o sobre el resultado de otras funciones. De esta manera, en un sistema de programación genética se definen dos conjuntos: el conjunto de las funciones y el conjunto de los terminales. En el árbol de la expresión, los terminales son las hojas mientras que las funciones son los nodos internos. Se puede consultar una definición más completa y formal de los conjuntos de funciones y terminales en [Koza (1992)].

El funcionamiento del operador de recombinación selecciona aleatoriamente un subárbol de cada padre, y posteriormente intercambia dichos subárboles dando lugar a dos nuevos individuos (los hijos). Hay definidas distintas variantes del operador de recombinación en programación genética [Banzhaf y otros (1998)].

También se han propuesto distintas variantes del operador de mutación [Banzhaf y otros (1998)], pero la definición más utilizada reemplaza un subárbol seleccionado aleatoriamente por otro generado de manera aleatoria. No obstante, Koza sugiere en [Koza (1992)] que no se utilice el operador de mutación.

2.2.2. Variantes de la programación genética

La programación genética propuesta por Koza ha dado lugar a múltiples variantes en los últimos años, como por ejemplo, la Evolución Gramatical, punto de partida de las extensiones desarrolladas en esta tesis. En la propuesta original de Koza, los programas se representan en forma de árbol, sin embargo, en los distintos sistemas aparecidos posteriormente, se plantean alternativas para la representación de los programas. En la actualidad, las distintas técnicas de programación genética se clasifican atendiendo a la representación que cada técnica utiliza pata los programas [Banzhaf y otros (1998)]. Se distinguen tres tipos:

- *Programación genética basada en árboles*: los programas se representan en forma de árboles.

- *Programación genética lineal*: los programas se representan linealmente como cadenas de bits/enteros o como sentencias de código.

- *Programación genética basada en grafos*: los programas se representan como grafos.

Aunque la clasificación anterior parece estar ampliamente extendida, en la bibliografía del área se pueden encontrar otras agrupaciones que atienden a criterios distintos de la representación de los programas. Por ejemplo, habitualmente se agrupan las técnicas de programación genética que utilizan gramáticas para garantizar la clausura sintáctica, en un grupo que se denomina "Programación genética basada en gramáticas" o "Programación genética gramatical". Obviamente, a este grupo pueden pertenecer sistemas que representan los programas en forma de árbol, así como sistemas que utilizan una representación lineal. Precisamente la Evolución Gramatical es una de las técnicas que pertenece, por una parte, a la categoría de "Programación genética lineal", y por otra al grupo de "Programación genética basada en gramáticas". Otro grupo de sistemas de programación genética que se puede encontrar en la literatura, es el formado por los sistemas que utilizan código máquina para representar los programas. A este grupo se le denomina "Programación genética en código máquina".

En los siguientes apartados se presenta una colección de distintas técnicas de programación genética encontradas en la revisión bibliográfica realizada en el contexto de esta tesis. El objetivo no es mostrar una lista completa de todas las propuestas encontradas en la literatura sino, dibujar el entorno en el que se ubica la Evolución Gramatical, punto de partida de esta tesis. La serie de técnicas que se describen, se agrupan atendiendo a la primera clasificación mostrada en este apartado, prestando especial atención a la programación genética lineal.

2.2.3. Programación genética basada en árboles

Además de la propuesta original de Koza, se puede encontrar en la literatura otros autores que también utilizan la representación de los individuos de la población en forma de árbol.

Whigham propone en su tesis doctoral [Whigham (1996)] incorporar a la programación genética de Koza el uso de gramáticas independientes del contexto para especificar el lenguaje de los programas (individuos de la población). Los programas se siguen representado en forma de árbol.

El sistema propuesto por Whigham también se puede clasificar dentro del grupo "Programación genética gramatical". El uso de la gramática junto con unos operadores genéticos basados en la misma, garantiza la corrección sintáctica de todos los individuos de la población, tanto en la población inicial como a lo largo del proceso evolutivo. El operador de recombinación intercambia subárboles que tengan como raíz el mismo no terminal de la gramática, asegurando la corrección sintáctica del nuevo árbol. El operador de mutación selecciona un no terminal del árbol y sustituye su subárbol por un nuevo subárbol que se genera utilizando la gramática.

Hörner [Hörner (1996)] realiza una propuesta similar a la de Whigham utilizando gramáticas independientes del contexto para la generación de los programas, y redefine de manera similar los operadores de recombinación y mutación. Junto con la propuesta teórica, Hörner presenta una librería de clases en C++ que denomina "Genetic Programming Kernel", que implementan el modelo.

2.2.4. Programación genética lineal

Dentro de esta categoría de programación genética se pueden distinguir dos enfoques diferentes. Por una parte, los sistemas en los que los programas se representan como secuencias de instrucciones de un lenguaje de programación imperativo o lenguaje máquina. Y por otra, los sistemas en los que los programas se representan como una cadena de bits, o de números enteros, junto con un mecanismo de transformación/traducción de dicha cadena en programas pertenecientes a un determinado lenguaje de programación. En este último enfoque, el espacio de genotipos no es el mismo que el de fenotipos, en oposición a los sistemas convencionales de programación genética basados en árboles en los que no se distingue entre estos dos espacios de forma explícita.

En la selección de sistemas de programación genética lineal que se describe a continuación, se dedicará especial atención a aquellos sistemas en los que existe una separación entre el espacio de genotipos y el de fenotipos ya que la Evolución Gramatical es uno de ellos.

Una de las primeras propuestas para evolucionar programas representados linealmente se debe a Cramer [Cramer (1985)]. Su principal objetivo era diseñar un lenguaje que fuera fácilmente manipulable por un algoritmo genético. Inspirado en el lenguaje de programación PL, Cramer propuso el

lenguaje JB que consta de cinco instrucciones cuya sintaxis y significado se describe a continuación.

(:BLOCK S_i S_j**)**	Evalúa secuencialmente las dos instrucciones S_i y S_j
(:LOOP V S**)**	Ejecuta V veces la instrucción S
(:SET V_n V_m**)**	Asigna a la variable V_n el valor de V_m
(:ZERO V**)**	Asigna a la variable V el valor 0
(:INC V**)**	Añade 1 a la variable V

Cada una de las instrucciones se puede representar por tres números naturales, el primero de ellos indentifica la instrucción y los dos siguientes los registros/variables implicados en la misma o bien referencias a otras instrucciones. Por lo tanto, cada programa se puede representar como una secuencia de números que se interpretan de la siguiente manera. Se divide la secuencia en instrucciones agrupando los números en grupos de tres. Si después de esta operación sobra algún número, se ignora. La primera instrucción se define como instrucción principal (MS, del inglés "Main Statement"), y el resto de instrucciones como instrucciones auxiliares (AS, del inglés "Auxiliary Statement"). Como ejemplo de codificación de las cinco instrucciones del lenguaje JB, en [Cramer (1985)] se presenta la siguiente lista:

(0 4 2)	→	**(:BLOCK** AS_4 AS_2**)**
(1 6 0)	→	**(:LOOP** V_6 AS_0**)**
(2 1 9)	→	**(:SET** V_1 V_9**)**
(3 17 8)	→	**(:ZERO** V_{17}**)** ;; el 8 se ignora
(4 0 5)	→	**(:INC** V_0**)** ;; el 5 se ignora

Los símbolos V_n representan variables, y AS_n instrucciones auxiliares. La secuencia de instrucciones se completa con un conjunto de variables internas inicializadas a 0 y un conjunto de la variables de entrada. Al final de la ejecución del programa, se puede devolver como salida cualquiera de las variables anteriores. Básicamente la ejecución consiste en la invocación de la instrucción principal (MS) que llamará a una o mas de las instrucciones auxiliares (AS). En [Cramer (1985)] se puede encontrar el siguiente ejemplo. La cadena de números naturales:

(0 0 1 3 5 8 1 3 2 1 4 3 4 5 9 9 2)

corresponde al siguiente programa en JB:

(0 0 1) → **(:BLOCK** AS_0 AS_1**)** ;; instrucción principal
(3 5 8) → **(:ZERO** V_5**)** ;; instrucción auxiliar 0
(1 3 2) → **(:LOOP** V_3 AS_2**)** ;; instrucción auxiliar 1
(1 4 3) → **(:LOOP** V_4 AS_3**)** ;; instrucción auxiliar 2
(4 5 9) → **(:INC** V_5**)** ;; instrucción auxiliar 3

En posteriores estudios, Cramer abandonó el lenguaje JB y lo sustituyó por TB, que utiliza una representación de los programas en forma de árbol. Hay autores que consideran que la propuesta de Cramer con su lenguaje TB junto con los primeros trabajos de Koza fueron el nacimiento de lo que posteriormente se ha denominado Programación Genética.

Posteriormente, Nordin propuso el sistema CGPS (del inglés "Compiling genetic programming system") en el que el proceso evolutivo se realiza sobre el propio código máquina [Nordin (1994)]. La motivación de su propuesta era conseguir eliminar el tiempo que se invierte en los sistemas de programación genética convencionales en interpretar los programas. Efectivamente, si los individuos/programas del sistema, en lugar de estar escritos en un lenguaje de programación, están ya disponibles en código máquina, su ejecución es inmediata y no requiere ningún tipo de traducción, lo que implica una importante mejora en términos de tiempo. Posteriormente, el sistema CGPS evolucionó convirtiéndose en AIM-GP (del inglés "Automatic induction of machine code by genetic programming") [Nordin (1997) y Nordin y otros (1999)]. En AIM-GP cada programa está expresado en código máquina, y los operadores de recombinación y mutación están diseñados específicamente para generar siempre individuos válidos. En [Nordin (1997)] se puede encontrar una descripción detallada del sistema. La empresa RML Technologies [http://www.rmltech.com/] implementó la versión comercial de la propuesta de Nordin bajo el nombre comercial Discipulus™.

Crepeau propuso GEMS (del inglés "Genetic Evolution of Machine Language Software") [Crepeau (1995)] uno de los sistemas más extensos para evolucionar código máquina. Su motivación era desarrollar un sistema que proporcionase unos conjuntos de funciones y terminales de una extensión considerablemente superior a los conjuntos de los sistemas que se habían desarrollado hasta el momento. El sistema integra un emulador del microprocesador Z-80 con más de 600 instrucciones. Cada instrucción se considera indivisible de manera que el operador de recombinación genera siempre individuos correctos.

Brameier y Banzhaf [Brameier y Banzhaf (2001)] proponen una nueva va-

riante de programación genética lineal en la que los programas que evolucionan se representan mediante instrucciones de código C. Los autores presentan en [Brameier y Banzhaf (2001)] la descripción del sistema y su aplicación en seis problemas de diagnóstico médico extraidos de la base de datos PROBEN1 [Prechelt (1994)]. Esta base de datos, originalmente desarrollada para probar sistemas de redes neuronales, contiene datos del mundo real de distintos ámbitos y un conjunto de indicaciones para realizar pruebas sobre los mismos. De esta manera, Bramier y Banzhaf, comparan el rendimiento de su método de programación genética lineal con el rendimiento de sistemas de redes neuronales, obteniendo resultados satisfactorios.

Los individuos se representan como una secuencia de instrucciones C pertenecientes a un conjunto reducido en el que hay operaciones aritméticas, sentencias condicionales y llamadas a funciones. El tamaño máximo de un individuo es de 256 instrucciones. En la tabla 2.10 se muestra el conjunto de instrucciones disponibles.

Tipo de instrucción	Notación
Operación aritmética	$v_i := v_j \;\; op \;\; v_k \mid c \quad op \in \{+, -, *, /\}$
Sentencia condicional	$if \;\; (v_i \;\; cmp \;\; v_k \mid c) \quad cmp \in \{>, \leq\}$
Llamada a función	$v_i := f(v_j) \quad f \in \{sin, cos, sqrt, exp, log\}$

Tabla 2.10: Instrucciones del lenguaje del sistema de programación genética lineal propuesto por Brameier y Banzhaf.

Puede observarse que las instrucciones trabajan con dos variables, o con una variable y una constante. Cada instrucción se representa por un vector de cuatro posiciones para almacenar, la identificación de la instrucción, los índices de las variables y opcionalmente el valor de la constante. Se reserva un byte para cada posición del vector, y por lo tanto, el número de variables se restringe a 256, y el valor de las constantes al intervalo $[0, 255]$. Dado que el número máximo de instrucciones de un individuo es 256, y cada instrucción ocupa 4 bytes, el tamaño máximo de un individuo es 1K, y por lo tanto, el sistema se puede considerar eficiente en términos de uso de memoria.

El operador de recombinación es de dos puntos e intercambia entre los padres segmentos situados en una posición aleatoria y de una longitud aleatoria. Los puntos del operador de recombinación siempre están situados entre dos instrucciones, nunca dentro de ellas para asegurar la clausura sintáctica. El operador de mutación trabaja dentro de las instrucciones sustituyendo

identificadores de instrucción, variables y constantes por otros valores pertenecientes a rangos válidos, asegurando también así la clausura sintáctica.

El sistema incorpora un algoritmo para eliminar código que no es efectivo, es decir, instrucciones que no tienen ningún efecto en el comportamiento del programa. Se puede establecer una analogía entre este tipo de instrucciones y los llamados *intrones* en biología molecular. La eliminación de este tipo de instrucciones se realiza antes de que el programa se ejecute, produciendo un considerable incremento en la velocidad de ejecución. Sin embargo, en realidad, las instrucciones no se borran del individuo, y se mantienen durante el proceso evolutivo. De manera semejante, en biología, los intrones se eliminan antes de la síntesis de proteínas, pero permanece en la cadena de ADN. Los autores, distinguen dos tipos de *intrones*. Los *intrones estructurales* son aquellas instrucciones en las que aparecen variables que no intervienen en los cálculos de las salidas del programa. En segundo lugar, los *intrones semánticos* corresponden a instrucciones o conjuntos de instrucciones que manipulan variables relevantes para las salidas del programa de manera que la manipulación mantiene inalterado el valor de la variable. Por ejemplo, una instrucción que suma a una variable el valor cero, o la secuencia de dos instrucciones de manera que en la primera se incrementa en uno el valor de una variable y en la segunda se decrementa. El algoritmo incorporado en el sistema únicamente elimina los intrones estructurales.

Otro sistema de programación genética lineal propuesto muy recientemente [Alonso y otros (2008)] utiliza para representar programas una estructura denominada *slp* (del inglés "straight line program"). Básicamente, un slp es una secuencia finita de instrucciones que tienen una estructura definida. A partir de un conjunto de funciones y un conjunto de terminales, cada instrucción es una asignación que tiene, en su parte izquierda, una variable, y en su parte derecha, una función que opera sobre terminales y sobre las variables definidas a la izquierda de las asignaciones de las instrucciones previas. Los autores denominan código no efectivo a cualquier instrucción cuya ejecución no tenga influencia en la salida del programa, y proponen la eliminación de dicho código antes de la evaluación del programa. El operador de recombinación se define de manera que intercambia bloques completos de cálculo de subexpresiones, entendiendo por subexpresión el conjunto de todas las instrucciones involucradas en la evaluación de una variable. El operador de mutación selecciona aleatoriamente una instrucción, y también de manera aleatoria uno de los argumentos de la función que aparece en la instrucción. Finalmente, el argumento seleccionado se sustituye por otro que también se elige aleatoriamente.

Banzhaf realizó una de las primeras propuestas para distinguir el espacio de búsqueda (genotipos) y el espacio de soluciones (fenotipos) [Banzhaf (1994)] en contraposición al único espacio existente en GP. Banzhaf propone que dado un algoritmo de búsqueda, se separen los espacios de búsqueda y de solución de manera que se mantenga una correspondencia entre ambos mediante una función que transforme los objetos/puntos del espacio de búsqueda en objetos/puntos del espacio de soluciones. Obviamente, este esquema admite cualquier algoritmo de búsqueda, como por ejemplo, un algoritmo evolutivo. Banzhaf sostiene que la no separación de ambos espacios hace que la resolución de cualquier problema se transforme en un problema de optimización "con restricciones". Considerando que las restricciones provienen de la propia representación de los individuos y de la definición de los operadores de recombinación y mutación. Esto hace que haya amplias zonas en el espacio de búsqueda (que es el mismo que el de las soluciones) que se convierten en inalcanzables, y como consecuencia de ello disminuye la potencia del algoritmo. Sin embargo, al separar los espacios, el espacio de búsqueda se compone de todos los genotipos (legales e ilegales) y los operadores no tienen restricciones ya que es la función de correspondencia entre ambos espacios la que transforma cada genotipo en un fenotipo correcto (sintácticamente correcto).

A continuación se describe brevemente el algoritmo propuesto por Banzhaf [Banzhaf (1994) y Keller y Banzhaf (1996)] denominado inicialmente BGP (del inglés "Binary genetic programming") y posteriormente renombrado DGP (del inglés "Developmental genetic programming"). Cada genotipo se representa con una cadena binaria y cada fenotipo es una cadena perteneciente a un determinado lenguaje. La función de correspondencia entre el espacio de genotipos y el de fenotipos es una "metáfora simplificada" del complicado proceso biológico de la síntesis de proteinas. La cadena binaria (genotipo) se considera formada por una serie de codones (cada codón corresponde a un determinado número de bits). Cada codón representa a un símbolo del lenguaje (de la misma manera que en biología molecular cada codón codifica un aminoácido). Además, esta codificación es redundante igual que en la naturaleza, es decir, un mismo símbolo del lenguaje puede estar representado por distintos codones, como un mismo aminoácido puede estar representado por distintos codones). La función de mapeo recorre la cadena binaria, agrupando los bits en sus correspondientes codones, y construyendo una cadena de símbolos del lenguaje aplicando la correspondencia previamente establecida entre condones y símbolos del lenguaje. Habitualmente esta cadena no es sintácticamente correcta, y se le aplica un mecanismo de reparación de acuerdo a la sintaxis del lenguaje.

De esta manera se obtiene una cadena de símbolos correcta. En concreto, este modelo solamente trabaja con lenguajes generados por gramáticas LALR(1) ya que el proceso de reparación se basa en las características de estas gramáticas. Posteriormente se edita la cadena sintácticamente correcta para incorporar información fija como puede ser una cabecera de función o un marco de programa principal. Por último, sólo queda compilar la secuencia editada para generar código ejecutable y así poder evaluar el *fitness* correspondiente.

En la misma época que Banzhaf, Paterson y Livesley propusieron el algoritmo GADS (del ingles "Genetic Algorithm for Deriving Software") [Paterson y Livesey (1996)]. En esta propuesta, los genes son números enteros y los fenotipos son programas escritos en un lenguaje de programación cuya sintaxis se representa en Forma Normal de Backus (BNF) [Alfonseca y otros (2007)]. Las producciones son numeradas, y por lo tanto, cada producción se puede representar por un número. Por último, la función de mapeo para generar los fenotipos a partir de los genotipos trabaja de la siguiente manera. Aunque los fenotipos se pueden generar como cadenas o como árboles, independientemente de la forma del fenotipo (cadena o árbol), el proceso de mapeo comienza inicializando el fenotipo con el axioma de la gramática y ordenando las reglas en una lista de forma que a cada una de ellas le corresponde un índice único. Posteriormente, se construye una derivación por la izquierda dirigida por los valores de los genes de la siguiente manera. Se leen los genes comenzando por la izquierda, y para cada gen se comprueba si su valor corresponde a una regla aplicable al no terminal actual (el situado más a la izquierda en la cadena que se esta derivando). Si la regla es aplicable se sustituye el no terminal actual por la parte derecha de la regla, y si no es aplicable se ignora el gen y se lee el siguiente. Cuando se han leído todos los genes, puede ocurrir que el fenotipo contenga no terminales. En ese caso, el fenotipo se puede reparar expandiendo los no terminales utilizando unos valores por defecto. Estos valores por defecto para cada no terminal se definen extendiendo la notación BNF. Los autores proponen otras alternativas para la gestión de fenotipos que contengan no terminales, como por ejemplo, rechazar el fenotipo o penalizarlo con el peor valor de *fitness*. Los autores reconocen que uno de los puntos débiles de su propuesta es la baja probabilidad de selección de cualquier regla para los casos de gramáticas extensas. Una de las soluciones que proponen para incrementar la probabilidad de determinadas reglas es duplicarlas dentro de la gramática. Otro de los puntos débiles de este sistema se debe a la manera en la que los genes son ignorados si no son aplicables al no terminal en proceso. Este método puede generar gran cantidad de intrones y un gran

número de fenotipos con no terminales en los que se aplicarían los valores por defecto definidos por el usuario.

Otro sistema muy similar a GADS es el propuesto por Freeman [Freeman (1998)], y que denominó CFG/GP (del inglés "Context Free Grammars GP"). La diferencia entre este sistema y GADS está en la selección del no terminal que se va a expandir con el gen actual. Mientras que en GADS se expande el no terminal situado más a la izquierda, en CFG/GP se puede expandir cualquier no terminal del fenotipo en construcción. Dada la similitud de ambos sistemas, los puntos débiles de ambos son similares.

La Evolución Gramatical propuesta por Michael O'Neill y Conor Ryan [Ryan y otros (1998), O'Neill (2001) y O'Neill y Ryan (2003)] tiene una gran similitud con los sistemas DGP y GADS. Los genotipos son cadenas binarias y los fenotipos programas escritos en un lenguaje de programación cuya sintaxis se representa en Forma Normal de Backus (BNF). En el capítulo 3 se describe detalladamente la Evolución Gramatical dado que requiere una atención especial por ser el punto de partida de las extensiones propuestas en esta tesis.

2.2.5. Programación genética basada en grafos

Teller y Veloso propusieron el sistema PADO (del inglés "Parallel Algorithm Discovery and Orchestration") [Teller y Veloso (1995b)] que trabaja con programas en lugar de las expresiones simbólicas propuestas por Koza. El objetivo principal de PADO es el reconocimiento se señales de cualquier tipo, como por ejemplo, imágenes. Un programa se representa mediante un grafo dirigido de N nodos. Cada nodo contiene una parte de acción y una parte de decisión de flujo. En cada programa hay nodos especiales, como son el nodo de inicio, el nodo de finalización, los nodos de llamadas a subprogramas y los nodos de llamadas a funciones de librería. Teller afirma que sus sistema es mejor clasificador de señales que los sistemas que utilizan expresiones simbólicas como por ejemplo la programación genética de Koza [Teller y Veloso (1995a)]. Además, la representación de los programas en forma de grafos dirigidos facilita la incorporación de bucles. En cuanto a los operadores de recombinación y mutación, el sistema PADO implementa un método de co-evolución que sus autores denominan operadores inteligentes.

Otros método de programación genética basado en grafos es "Cellular encoding", propuesto por Gruau [Gruau (1994)]. El objetivo del sistema es el diseño automático de la estructura de redes neuronales. Para ello, las redes se

representan en forma de grafo y son evolucionadas utilizando programación genética.

2.2.6. Otros sistemas

Es conveniente mencionar otros sistemas que resultan más difíciles de clasificar en los grupos que se han descrito anteriormente, en particular aquellos que se basan o utilizan en algún sentido lenguajes de programación lógica (Prolog) o resolvedores similares. En [Christiansen (1990)] se describe la gran cantidad de retroseguimiento (*backtracking*) y la posibilidad de bucles infinitos inherentes a la estrategia subyacente de resolución de los sistemas lógicos como Prolog así como la semidecibilidad teórica de la lógica de primer orden. Este inconveniente podría hacer que los sistemas basados en Prolog sean descartados, en general, para los experimentos de programación genética ya que en la mayoría de ellos el rendimiento es importante.

2.3. Programación genética y semántica

La utilización de información semántica en programación genética es un área de investigación que parece que está creciendo en los últimos años. Los trabajos enmarcados en el área que se encuentran en la bibliografía se pueden clasificar, como se sugiere en [Uy y otros (2009b)], en función del mecanismo utilizado para la expresión de la semántica, distinguiéndose tres grupos:

- Métodos que utilizan gramáticas.
- Técnicas basadas en métodos formales.
- Métodos que modifican el funcionamiento de los operadores genéticos.

En los siguientes apartados se describen algunos de los trabajos encontrados en la revisión bibliográfica.

2.3.1. Métodos que utilizan gramáticas

En esta categoría de la clasificación se ubican los dos métodos propuestos en esta tesis. El primero de ellos [de˜la Cruz y otros (2005)] utiliza gramáticas de atributos para la gestión de información semántica de los individuos de la población y el segundo [Ortega y otros (2007)] hace uso de gramáticas de Christiansen.

En la revisión bibliográfica se ha encontrado un trabajo en el que se resuelve el problema de la mochila entera 0/1 con múltiples mochilas utilizando gramáticas de atributos junto con Evolución Gramatical [O'Neill y otros (2004) y Cleary y O'Neill (2005)]. En este trabajo no se propone un método general sino únicamente la resolución del problema de la mochila, diseñando una gramática de atributos específica para el mismo. Los autores concluyen que los resultados obtenidos para el problema concreto de la mochila muestran una clara mejora en el rendimiento de Evolución Gramatical cuando se incorporan aspectos semánticos con gramáticas de atributos.

Paterson en su tesis doctoral [Paterson (2002)] estudia la combinación del algoritmo GADS propuesto por él mismo en [Paterson y Livesey (1996)] y gramáticas de atributos reflectivas para la generación de fenotipos en el lenguaje de programación S-Algol con el objetivo de que los fenotipos, además de ser correctos sintácticamente, también sean correctos en cuanto a la comprobación de tipos. Una gramática de atributos reflectiva es una gramática de atributos en la que en la expansión de un símbolo no terminal, la propia gramática es tratada como un atributo heredado de manera similar a las gramáticas de Christiansen. La gran diferencia entre ambos tipos de gramáticas es que las de Chrsitiansen han sido formalizadas por su autor, y objeto de investigación desde que fueron formuladas en 1985 por Christiansen.

2.3.2. Técnicas basadas en métodos formales

Algunos autores han observado dificultades en el diseño de las funciones de adecuación (*fitness*) frecuentes en diferentes algoritmos de programación genética. [Johnson (2002b)] observa que estas funciones generalmente se basan en la ejecución del programa evolucionado y la aplicación de una métrica que mida la distancia entre la solución deseada (objetivo) y la obtenida. Consideran general el uso de un pequeño conjunto de valores para probar los programas que pueden plantear riesgos de sobre ajuste y mal comportamiento del código generado para otros valores. En [Johnson (2002a)] describen, como posible solución, una serie de técnicas provenientes de la informática teórica y que se aplican para obtener información de los programas sin necesidad de ejecutarlos (medidas de su complejidad tanto temporal como en función del consumo de espacio, número de posibles ramas de ejecución, propiedades expresadas como condiciones sobre el valor de sus variables, etc...) En otros trabajos el mismo autor experimenta con algunas de estas técnicas e intenta probar su viabilidad y potencia en algunos problemas concretos. En [Johnson (2002b)] aborda el uso técnicas de análisis estático. Estas técnicas permiten afirmar de los programas, sin necesidad de ejecutarlos, que cier-

tas propiedades se cumplen independientemente de los valores concretos que tomen como entrada. Estas propiedades pueden referirse a su rendimiento (consumo de memoria, número de posibles ramas de ejecución) o a su comportamiento (por ejemplo, el rango de valores que sus variables pueden tomar). El análisis estático se basa en la abstracción de los operadores presentes en los programas analizados. Sólo se tienen en cuenta las propiedades de interés. La ejecución del programa concreto se sustituye por su expresión como secuencia de operaciones abstractas. La evaluación de estas expresiones abstractas genera el resultado del análisis estático. En este trabajo se describe de una forma teórica y general la manera en la que, dependiendo del tipo de problema, pueden aplicarse estas técnicas. En el mejor de los casos la función de adecuación podrá expresarse completamente como una lista de condiciones cada una de las cuales se podrá comprobar mediante análisis estático. En otros casos, ciertas condiciones se comprobarán mediante el análisis estático pero otras tendrán que ser expresadas de otra forma e integradas en una función de *fitness* global mediante técnicas multiobjetivo o multicriterio. Para ilustrar la viabilidad y potencia de este enfoque los autores proponen un problema del primer tipo (su función de adecuación se puede expresar por completo mediante análisis estático): encontrar en el interior de un rectángulo una disposición geométrica que cumpla una serie de restricciones de solapamiento para un conjunto de rectángulos. Este problema se puede considerar una abstracción del algoritmo que utilizan los interfaces de las aplicaciones gráficas cuando de manera automática van colocando en la pantalla el ordenador las ventanas que se van creando. Se aborda su diseño automático mediante EG. Las restricciones que debe cumplir el algoritmo que coloca las ventanas pueden expresarse por completo como una serie de ecuaciones sobre los valores máximos y mínimos de las dos dimensiones de los rectángulos. Estos valores pueden ser calculados mediante análisis estático sin necesidad de ejecutar el programa. La función de *fitness* simplemente cuenta el número de ecuaciones satisfechas y el mejor individuo es el que más satisface. En [Johnson (2007)]los mismos autores se centran en el uso de técnicas de *model checking*. Estas técnicas utilizan algún tipo de lógica temporal para enunciar ciertas propiedades que los programas deben cumplir a medida que el tiempo avanza a lo largo de la ejecución del programa. Como sistema de *model checking* utilizan CTL (*Computational Tree Logic*, http://www.cs.cmu.edu/TILDEmodelcheck/smv.html) y la versión del Sttutgart Model-Checking Kit el algoritmo SMV (http://www.fmi.uni-sttutgart.de/szs/tools/mckit/). El problema que abordan para ilustrar la potencia y viabilidad de este enfoque es el diseño de máquinas de estado finito (en concreto expendedores de bebidas calientes). El comportamiento de la máquina objetivo puede expresarse por completo como una serie de propo-

siciones expresadas en la lógica temporal que utilizan. De nuevo la función de adecuación tiene en cuenta el número de proposiciones satisfechas por los programas evolucionados.

Es importante resaltar que este enfoque no supone una modificación estructural a los algoritmos de programación genética ya que el lugar en el que más frecuentemente, en algunos algoritmos es el único posible, se pueden describir las condiciones que se pide a un programa para considerarlo solución es en la función de *fitness*.

Queda claro que, aunque de indudable interés, estas técnicas difieren de nuestros objetivos ya que lo que proponemos son modelos que incorporen en el proceso de generación de los individuos (y no en el de su evaluación) el uso de representaciones formales sofisticadas y capaces de expresar cualquier tipo de restricción.

2.3.3. Métodos que modifican el funcionamiento de los operadores genéticos

Los métodos que utilizan los operadores genéticos para incorporar información semántica proponen variantes de los operadores genéticos estándar. En la revisión bibliográfica se han encontrado propuestas relativas al operador de recombinación, y otras que redefinen el operador de mutación. En los párrafos siguientes se muestran, por orden cronológico, los métodos encontrados.

Majeed y Ryan describen en [Majeed y Ryan (2006a)] un nuevo operador de recombinación que trabaja buscando todos los posibles contextos en los que se puede ubicar el subárbol que se va a intercambiar en una operación de recombinación, y evaluándolos para poder seleccionar el mejor contexto destino. De esta manera, se consigue una recombinación menos destructiva. El funcionamiento del operador es el siguiente. Partiendo de dos padres, P1 y P2, se selecciona aleatoriamente un punto de recombinación en el padre P2. El siguiente paso consiste en buscar en P1 todos los posibles puntos de recombinación que generarían un hijo válido. La búsqueda de todos los posibles puntos en P1, asegura la identificación del mejor contexto para el subárbol de P2 dentro de P1. A continuación, se construyen todos los posibles hijos (uno por cada punto de recombinación) y se evalúan. El hijo que obtenga una mejor evaluación es el que pasa a formar parte de la siguiente generación. Se repite el mismo proceso para un subárbol aleatorio del padre P1. Según sus autores, este nuevo operador puede considerarse como un operador de búsqueda local, ya que, realiza una búsqueda por

fuerza bruta para encontrar el mejor contexto de un subárbol. La selección del mejor contexto, en oposición al contexto aleatorio que utiliza el operador de recombinación estándar, causa saltos más pequeños dentro del espacio de búsqueda. Dado que en [Banzhaf y otros (1998)] está demostrado que el operador de recombinación estándar es más efectivo al comienzo de la búsqueda porque es un operador de búsqueda global, se puede deducir que al final del proceso, el operador estándar no es una buena opción, mientras que sí lo es un operador de "refinamiento". Por ello, los autores proponen el uso del operador estándar en las primeras fases de la ejecución, y la utilización del nuevo operador en la parte final. Los resultados obtenidos con el uso del nuevo operador en diferentes problemas permite concluir a los autores que dicho uso mejora el *fitness* medio de la población y permite controlar el crecimiento de los individuos a lo largo del proceso evolutivo. En cuanto al número de individuos evaluados, el nuevo operador es capaz de generar mejores individuos con menos evaluaciones porque permite trabajar con poblaciones mucho más pequeñas. Se puede encontrar una descripción detallada de las mejoras que proporciona el nuevo operador en [Majeed y Ryan (2006b)].

Majeed y Ryan describen en [Majeed y Ryan (2007)]una variante del operador de mutación estándar que denominan *Context-Aware mutation*. Este nuevo operador de mutación reemplaza los subárboles por otros seleccionados de un repositorio en vez de ser generados de manera aleatoria como ocurre con el operador de mutación estándar. Este repositorio se crea ejecutando una vez el experiemento, y cogiendo de la última generación los individuos cuyo *fitness* es superior al *fitness* medio. De esta manera, la realización de un experimento consiste en, una primera ejecución en la que se genera el repositorio de subárboles, seguida de tantas ejecuciones como se quiera, en las que se utiliza el repositorio. Comparando con los operadores estándar, se mejoran los resultados cuando en la ejecución en la que se crea el repositorio se usa el operador de recombinación *Contest-Aware crossover* propuesto por los mismos autores y en el resto de ejecuciones se utiliza el nuevo operador de mutación.

Beadle y Johnson proponen en [Beadle y Johnson (2008)] un nuevo algoritmo llamado *Semantically Driven Crossover* (SDC) aplicable a la resolución de problemas sobre funciones lógicas. La idea principal del algoritmo es la representación canónica de los individuos de la población mediante los diagramas *Reduced ordered binary decision diagramas* (ROBDDs). Esta representación permite comprobar la equivalencia semántica de dos individuos de manera que, si dos programas se reducen al mismo ROBDD

son considerados semánticamente eqivalentes. El algoritmo SDC utiliza el operador de recombinación estándar de Koza hasta que se consiguen generar programas hijos que no sean semánticamente equivalentes a sus padres, es decir, si la recombinación de los padres genera hijos semánticamente equivalentes a los padres, son rechazados y no entran en la población. Según sus autores los principales resultados obtenidos con este nuevo algoritmo son el aumento del rendimiento debido al incremento de la diversidad genética de la población, y la disminución del tamaño de los individuos.

Inspirados en [Beadle y Johnson (2008)], los autores de [Uy y otros (2009a)] proponen un operador de recombinación que denominan *Semantic Aware Crossover* (SAC) que trabaja en función de la equivalencia semántica de subárboles. La equivalencia semántica de dos subárboles se determina evaluando las expresiones que representan en un conjunto aleatorio de puntos. Se considera que los subárboles son semánticamente equivalentes si las evaluaciones obtenidas en dichos puntos están suficientemente cerca (en función de un parámetro denominado 'sensibilidad semántica").

El funcionamiento del operador SAC es el siguiente: en cada operación de recombinación, se evalúa la equivalencia semántica de los subárboles cuyas raíces coinciden con los puntos de recombinación. Solamente se ejecuta la recombinación si los dos subárboles no son semánticamente equivalentes. A partir de los resultados obtenidos con el nuevo operador en el problema de regresión simbólica de funciones reales, los autores afirman que el operador SAC mejora el rendimiento del operador sin semántica, tanto en frecuencia acumulada de éxito como en mejor *fitness* medio.

También proponen el uso del concepto de equivalencia semántica para ser aplicado en árboles completos, es decir, en individuos de la población, de la siguiente manera. A partir de dos padres, se generan dos hijos aplicando el operador de recombinación SAC. Si los hijos no son semánticamente equivalentes a sus padres, se incorporan a la población de la siguiente generación, y si lo son, entonces los padres pasan a la siguiente generación en lugar de sus hijos.

Beadle y Johnson proponen en [Beadle y Johnson (2009)] un nuevo algoritmo llamado *Semantically Driven Mutation* (SDM) aplicable a la resolución de problemas sobre funciones lógicas. De la misma manera que en el algoritmo *Semantically Driven Crossover* (SDC), porpuesto por llos mismo autores, la idea principal del algoritmo es la representación canónica de los individuos de la población mediante los diagramas *Reduced ordered binary decision diagramas* (ROBDDs). Esta representación permite comprobar la equivalencia semántica de dos individuos de manera que, si dos programas

se reducen al mismo ROBDD son considerados semánticamente eqivalentes. El algoritmo SDM aplica un número prefijado de veces el operador de mutación estándar de Koza sobre cada individuo sujeto a mutación hasta conseguir que el programa mutado no sea semánticamente equivalente al programa original. Los autores resuelven siete problemas con funciones booleanas y prueban que el nuevo algoritmo SDM aumenta el rendimiento en comparación con el operador de mutación estándar.

Capítulo 3

Evolución gramatical

La Evolución Gramatical, a partir de ahora EG, propuesta por Michael O'Neill y Conor Ryan en [Ryan y otros (1998), O'Neill (2001) y O'Neill y Ryan (2003)] como algoritmo de programación automática evolutiva, es uno de los dos pilares de esta tesis. El segundo punto de apoyo son las representaciones formales completas de los lenguajes de programación de alto nivel, como las gramáticas de atributos y las gramáticas de Christiansen.

En este capítulo se presenta de manera detallada el modelo de EG haciendo una descripción general del método y dedicando una atención especial al algoritmo de transformación de genotipo a fenotipo.

3.1. Descripción general

EG es un algoritmo evolutivo capaz de generar automáticamente programas escritos en cualquier lenguaje de programación. El proceso evolutivo no se realiza sobre los programas sino sobre cadenas binarias de longitud variable (genotipos). Los programas (fenotipos) se generan mediante un proceso de transformación en el que las cadenas binarias se utilizan para seleccionar reglas de producción de una gramática representada en Forma Normal de Backus (BNF) [Alfonseca y otros (2007)]. Este mecanismo de construcción de programas permite asegurar que siempre son sintácticamente correctos. Una vez generados los programas, una función de *fitness* evalúa su adecuación a la solución del problema. La independencia entre el algoritmo evolutivo y los programas generados proporcionada por el proceso de transformación de genotipo a fenotipo permite que EG se beneficie de todos los avances en el área de los algoritmos evolutivos. Por otra parte, el otro módulo del sistema EG, la gramática representada en BNF define el lenguaje de los programas generados permitiendo así evolucionar programas en cualquier lenguaje.

La separación entre el espacio de búsqueda (genotipos) y el espacio de soluciones (fenotipos o programas) permite que no exista ninguna restricción sobre los genotipos y por lo tanto tampoco sobre la búsqueda en su correspondiente espacio, manteniéndose en paralelo la validez de las soluciones [Keller y Banzhaf (1996)]. En [Yu y Bentley (1998)] se puede encontrar una descripción de distintos métodos de generación de fenotipos válidos a partir de genotipos. Otro de los beneficios del proceso de transformación de genotipo en fenotipo es que permite simular el fenómeno biológico habitualmente referido como degeneración del código genético consistente en la correspondencia entre distintos codones y el mismo aminoácido. Este fenómeno permite la ocurrencia de mutaciones silenciosas, que son aquellas que no tienen efecto en el fenotipo del individuo y que son la base de la teoría neutral de la evolución que formuló Kimura [Kimura (1983)].

EG utiliza como método de búsqueda de la solución un algoritmo evolutivo, en concreto, un algoritmo genético, pero sería válido cualquier método de búsqueda capaz de trabajar con cadenas binarias o enteras de longitud variable [O'Sullivan y Ryan (2002)]. Según la propuesta original, utilizando un algoritmo genético como método de búsqueda, el esquema básico de un sistema de evolución gramatical es el siguiente:

- Se inicializa la población, los genotipos de sus individuos, de manera aleatoria ; cada genotipo es una cadena binaria de una longitud inicialmente limitada pero que variará en el proceso evolutivo.

- A partir de cada genotipo se genera un fenotipo con un algoritmo de transformación que se describe en detalle posteriormente.

- Los fenotipos de toda la población se evalúan con una función de *fitness* que cuantifica el acercamiento a la solución del problema que se esté tratando.

- Una vez que está inicializada la población, comienza el proceso de evolución en el que, en cada paso evolutivo, se seleccionan los mejores individuos (aquellos con mejor función de *fitness*) para que actúen como padres y generen una progenie que sustituirá a los perores individuos de la población (aquellos con peor *fitness*). La creación de los hijos a partir de los padres se realiza con los genotipos, aplicando los operadores de recombinación de un punto y mutación propios de los algoritmos genéticos. Los autores proponen también el uso de un operador de duplicación de material genético.

- El proceso termina cuando se encuentra una solución que se considera aceptable, o se alcanza algún límite preestablecido como por ejemplo

un número de generación, un número de ejecuciones de la función de *fitness*, un tiempo de proceso, etc

La técnica EG, según la plantean sus creadores, tiene una característica muy importante, **la gramática es un componente del sistema**, y por lo tanto se puede sustituir de una manera muy sencilla. Esta característica de la técnica permite generar soluciones al problema que se esté tratando, en distintos lenguajes sin ningún coste/esfuerzo añadido.

3.2. Algoritmo de transformación de genotipo a fenotipo

Uno de los componentes de la técnica de EG es la gramática independiente del contexto del lenguaje de soluciones posibles del problema que se pretende resolver.

El algoritmo de transformación de genotipo a fenotipo consiste básicamente en construir, a partir del axioma de la gramática, una derivación por la izquierda del fenotipo. La manera de seleccionar la regla que se va a utilizar se explica con detalle más adelante. La transformación, por lo tanto, comienza con el axioma de la gramática y el genotipo, que es leído en grupos de 8 bits para generar los codones. En cada paso de la derivación, un codón determina la regla aplicada al símbolo no terminal más a la izquierda del fenotipo en construcción. La alternativa sugerida inicialmente por Ryan y O'Neill para la selección de una regla a partir del valor de un codón, es utilizar el módulo del codón con el número de reglas aplicables al no terminal que se está derivando. Por ejemplo, supóngase una gramática que tiene un símbolo no terminal *op* con cuatro reglas:

$$
\begin{array}{lllll}
(0) & <op> & \rightarrow & + \\
(1) & & | & - \\
(2) & & | & / \\
(3) & & | & *
\end{array}
$$

Si en un instante de la ejecución del algoritmo de transformación de genotipo a fenotipo el valor del codón en proceso es 49, y el símbolo no terminal más a la izquierda es *op*, entonces se aplicaría la regla (1) del no terminal *op* ya que $1 = 49 \ mod \ 4$.

Los codones se utilizan en orden secuencial, comenzando por el primero, es decir, en general, el primer codón es el que se utiliza para decidir cuál de

las producciones del axioma se va a utilizar en la primera derivación del algoritmo, el segundo codón permite seleccionar la regla del no terminal situado más a la izquierda en el segundo paso del algoritmo y así sucesivamente. En el caso particular de los símbolos no terminales que solamente tienen una regla, no tiene sentido el concepto de selección de regla ya que únicamente hay una alternativa. Por lo tanto, en estos casos no se utiliza ningún codón del genotipo.

Durante el proceso de traducción de genotipo a fenotipo, puede ocurrir que se agoten los codones del genotipo antes de que el fenotipo esté completamente terminado, es decir, esté formado únicamente por símbolos terminales de la gramática. Para estos casos los autores del modelo de evolución gramatical proponen un operador (*wrapping*) cuya funcionalidad consiste en reutilizar los codones para continuar el proceso de traducción. La reutilización se implementa volviendo a utilizar el genotipo desde su primer codón una vez que se haya consumido el último. Este operador se inspira en el mecanismo de solapamiento de genes que ocurre en algunos organismos como por ejemplo las bacterias [Lewin (1999)]. Si ocurriese que el fenotipo no se termina de construir correctamente incluso después de aplicar una o más veces el nuevo operador, se le asignaría al individuo el peor valor de fitness y los mecanismos de selección y reemplazamiento se encargarían de hacer desaparecer ese individuo de la población.

El proceso de transformación de genotipo a fenotipo termina cuando en uno de los pasos del algoritmo, el fenotipo en construcción pertenece al lenguaje generado por la gramática, es decir, está formado únicamente por símbolos terminales. No obstante, puede ocurrir, que se encuentre el final del genotipo y el fenotipo en construcción todavía contenga símbolos no terminales de la gramática, en ese caso, se reutilizaría la información genética (operador *wrapping*) un número máximo de veces. Si alcanzada esa cota superior de número de veces que se reutiliza la información genética, no se ha obtenido un fenotipo correcto (sólo con símbolos terminales) se considera terminado el proceso y se le asigna al individuo el peor valor de fitness.

Para clarificar el algoritmo de transformación de genotipo a fenotipo se utilizará un ejemplo propuesto en [O'Neill y Ryan (2003)] en el que la gramática independiente del contexto del sistema es $G = \{\Sigma_N, \Sigma_T, P, S\}$, donde el conjunto de símbolos no terminales Σ_N es,

$$\Sigma_N = \{expr, op, pre\_op, var\}$$

el conjunto Σ_T de terminales es,

$$\Sigma_T = \{\sin, +, -, /, *, x, 1{,}0, (,)\}$$

y el axioma,

$$S = expr$$

Por último, el conjunto de reglas P contiene los siguientes elementos:

(0) <*expr*> → <*expr*><*op*><*expr*>
(1) | (<*expr*><*op*><*expr*>)
(2) | <*pre_op*> (<*expr*>)
(3) | <*var*>

(0) <*op*> → +
(1) | −
(2) | /
(3) | ∗

(0) <*pre_op*> → sin

(0) <*var*> → *x*
(1) | 1,0

En general se numeran de manera independiente las reglas de cada símbolo no terminal para facilitar el seguimiento del algoritmo. Recuérdese que en cada paso, la regla seleccionada para derivar el no terminal situado más a la izquierda, se obtiene haciendo el módulo entre el codón actual y el número de reglas del no terminal, por lo tanto, el valor obtenido con esta operación, indica directamente la regla seleccionada en la numeración independiente que se emplea en las reglas de cada símbolo no terminal.

En las dos tablas siguientes se muestra el genotipo que se va a utilizar en el ejemplo que nos ocupa. Por razones de espacio se ha dividido el genotipo en dos tablas, correspondiendo la primera de ellas al principio del genotipo, y la segunda a la parte final del mismo.

220	40	16	203	101	53	202	203	102	55	220

202	19	130	37	202	203	32	39	202	203	102

A continuación se describe la traza de la ejecución completa del algoritmo. Se comienza inicializando el fenotipo con el axioma de la gramática. En

este momento inicial, el símbolo más a la izquierda que se va a derivar es precisamente el axioma, *expr*. Las reglas que se pueden utilizar para derivar el axioma son las cuatro siguientes:

(0) <*expr*> → <*expr*><*op*><*expr*>
(1) | (<*expr*><*op*><*expr*>)
(2) | <*pre_op*> (<*expr*>)
(3) | <*var*>

Para seleccionar una de las cuatro reglas se utiliza el primer codón del genotipo, que es 220 y se realiza la operación módulo entre el valor del codón y el número de reglas del no terminal que se está derivando, que en este caso es 4, obteniéndose 220 mod 4 = 0. Es decir, se utiliza la regla número 0 del símbolo no terminal *expr*. Al sustituir en el fenotipo en construcción el axioma por su regla seleccionada, se obtiene la siguiente forma sentencial:

<*expr*><*op*><*expr*>

En el siguiente paso del algoritmo, el primer no terminal más a la izquierda vuelve a ser *expr* y el siguiente codón del genotipo vale 40, de manera que permite seleccionar para la derivación la regla 0 ya que 40 mod 4 = 0, y el fenotipo en construcción pasaría a ser el siguiente:

<*expr*><*op*><*expr*><*op*><*expr*>

El mismo proceso ocurre con el siguiente codón, cuyo valor es 16, dando lugar a la forma sentencial:

<*expr*><*op*><*expr*><*op*><*expr*><*op*><*expr*>

El algoritmo continúa con el codón 203 y el símbolo *expr* situado más a la izquierda, que es sustituido esta vez por el no terminal *var* que es su cuarta parte derecha ya que 203 mod 4 = 3 obteniéndose la siguiente forma sentencial:

<*var*><*op*><*expr*><*op*><*expr*><*op*><*expr*>

En este instante, el símbolo situado más a la izquierda es *var*. Las reglas de este no terminal son dos:

(0) <*var*> → *x*
(1) | 1,0

El codón actual es 101 y como 101 $mod\ 2 = 1$ se selecciona la parte derecha de la segunda regla, es decir, se sustituye el símbolo *var* por el terminal 1,0 quedando el fenotipo en construcción de la siguiente manera:

1,0 <*op*><*expr*><*op*><*expr*><*op*><*expr*>

El próximo símbolo no terminal situado más a la izquierda es *op*, y el codón actual vale 53, por lo tanto, se selecciona la regla del operador de resta, dando lugar a la siguiente forma sentencial:

1,0− <*expr*><*op*><*expr*><*op*><*expr*>

De nuevo, se deriva el símbolo no terminal *expr* haciendo uso del siguiente codón (202). La operación módulo indica la selección de la tercera regla del símbolo, y el fenotipo en construcción queda de la siguiente manera:

1,0− <*pre_op*> (<*expr*>) <*op*><*expr*><*op*><*expr*>

Ahora, el no terminal más a la izquierda que se deriva es *pre_op*, pero este símbolo solamente tiene una parte derecha, y por lo tanto no hay opción de elegir una regla para la derivación, lo que elimina la necesidad de utilizar un codón como mecanismo de selección de regla. Es decir, no se consume ningún codón. El fenotipo actual sería:

1,0 − sin(<*expr*>) <*op*><*expr*><*op*><*expr*>

El proceso se repite hasta que ocurre una de las dos siguientes situaciones:

- se obtiene una cadena de terminales y, por tanto, un fenotipo correcto.
- se consume el genotipo (incluyendo la aplicación del operador *wrapping*) sin eliminar todos los no terminales y, por tanto, sin obtener un fenotipo correcto.

En la tabla 3.1 se muestra la ejecución completa paso a paso del algoritmo de transformación de genotipo a fenotipo. La primera columna hace referencia al paso del algoritmo, la segunda contiene el valor del codón que selecciona la regla para expandir el no terminal situado más a la izquierda en el fenotipo en construcción que se muestra en la tercera columna. En los pasos 8, 14, 19 y 24 del algoritmo, el no terminal situado más a la izquierda es *pre_op*, y como este símbolo sólo tiene una parte derecha no se consume ningún codón en el paso y, por ello, aparece vacía la segunda columna de la tabla. Como puede observarse, el algoritmo termina en 26 pasos, utilizando exactamente los 22

codones del genotipo. Podría haber ocurrido que no se hubieran utilizado todos los codones, es decir, que el genotipo estuviera terminado con menos de 22 codones. También podría haber pasado que después de aplicarse las reglas de la gramática determinadas por los 22 valores, el fenotipo contuviese todavía símbolos no terminales. En el primer caso simplemente no se utilizan los últimos codones, y en el segundo se puede aplicar el operador de *wrapping* tantas veces como se considere necesario.

paso del algoritmo	valor codón	fenotipo actual
1	220	<*expr*>
2	40	<*expr*><*op*><*expr*>
3	16	<*expr*><*op*><*expr*><*op*><*expr*>
4	203	<*expr*><*op*><*expr*><*op*><*expr*><*op*><*expr*>
5	101	<*var*><*op*><*expr*><*op*><*expr*><*op*><*expr*>
6	53	1,0 <*op*><*expr*><*op*><*expr*><*op*><*expr*>
7	202	1,0− <*expr*><*op*><*expr*><*op*><*expr*>
8		1,0− <*pre_op*> (<*expr*>) <*op*><*expr*><*op*><*expr*>
9	203	$1{,}0 - \sin(<expr>)$ <*op*><*expr*><*op*><*expr*>
10	102	$1{,}0 - \sin(<var>)$ <*op*><*expr*><*op*><*expr*>
11	55	$1{,}0 - \sin(x)$ <*op*><*expr*><*op*><*expr*>
12	220	$1{,}0 - \sin(x) *$ <*expr*><*op*><*expr*>
13	202	$1{,}0 - \sin(x) *$ <*expr*><*op*><*expr*><*op*><*expr*>
14		$1{,}0 - \sin(x) *$ <*pre_op*> (<*expr*>) <*op*><*expr*><*op*><*expr*>
15	19	$1{,}0 - \sin(x) * \sin(<expr>)$ <*op*><*expr*><*op*><*expr*>
16	130	$1{,}0 - \sin(x) * \sin(<var>)$ <*op*><*expr*><*op*><*expr*>
17	37	$1{,}0 - \sin(x) * \sin(x)$ <*op*><*expr*><*op*><*expr*>
18	202	$1{,}0 - \sin(x) * \sin(x) -$ <*expr*><*op*><*expr*>
19		$1{,}0 - \sin(x) * \sin(x) -$ <*pre_op*> (<*expr*>) <*op*><*expr*>
20	203	$1{,}0 - \sin(x) * \sin(x) - \sin(<expr>)$ <*op*><*expr*>
21	32	$1{,}0 - \sin(x) * \sin(x) - \sin(<var>)$ <*op*><*expr*>
22	39	$1{,}0 - \sin(x) * \sin(x) - \sin(x)$ <*op*><*expr*>
23	202	$1{,}0 - \sin(x) * \sin(x) - \sin(x) *$ <*expr*>
24		$1{,}0 - \sin(x) * \sin(x) - \sin(x) *$ <*pre_op*> (<*expr*>)
25	203	$1{,}0 - \sin(x) * \sin(x) - \sin(x) * \sin(<expr>)$
26	102	$1{,}0 - \sin(x) * \sin(x) - \sin(x) * \sin(<var>)$
		$1{,}0 - \sin(x) * \sin(x) - \sin(x) * \sin(x)$

Tabla 3.1: Traza del algoritmo de transformación de genotipo a fenotipo

Capítulo 4

Ajuste de evolución gramatical

Dado que EG es el punto de partida de las propuestas presentadas en esta tesis, el primer objetivo fue reproducir los experimentos propuestos en [O'Neill y Ryan (2003)] por los creadores de la técnica. La idea fundamental era asegurar que la implementación desarrollada de EG trabajaba correctamente, es decir, permitía obtener resultados similares o comparables a los que aparecen en [O'Neill y Ryan (2003)]. Una vez asegurado ese buen funcionamiento ya era posible abordar el segundo y principal objetivo, la comparación de resultados de experimentos realizados con las dos técnicas EG y EG con semántica.

En el proceso de intentar reproducir los experimentos antes mencionados, fueron necesarios ciertos ajustes y modificaciones del algoritmo original de EG. En este capítulo se detallan los ajustes realizados.

Veo necesario indicar previamente una consideración de vital importancia para comprender las discusiones sobre el rendimiento que incluye este capítulo. Hemos encontrado muchas dificultades para comprender las curvas de rendimiento que los autores del modelo de Evolución Gramatical muestran en sus artículos. La existencia de discrepancias entre diferentes versiones de aparentemente las mismas gráficas y, sobre todo, con las fuentes que citan, nos hacen concluir que la medida del rendimiento que aparece en las gráficas de los autores de EG no es la probabilidad acumulada de éxito sino la probabilidad de encontrar al menos una solución en un determinado número de ejecuciones según se describe en [Koza (1992)]. Sin embargo, es práctica habitual a la hora de evaluar los rendimientos de diferentes algoritmos de programación automática evolutiva (tal y como los autores de GE declaran explícitamente en sus publicaciones) comparar sus probabilidades acumuladas de éxito. Ese fue nuestro objetivo inicial. La naturaleza de las diferentes medidas hace imposible obtener ninguna conclusión (respecto a la mejoría o no en el rendimiento) excepto si se utiliza la misma.

4.1. Codificación del genotipo

La codificación del genotipo que se utiliza en EG original es una cadena de bits. Esa cadena se traduce a una serie de números enteros de manera que cada número entero corresponde a una secuencia de 8 bits. Los autores de EG establecen una analogía entre la cadena de bits y la cadena de doble hélice de ADN y hacen corresponder el proceso de escaneo de bits para obtener enteros con el proceso biológico de transcripción del ADN en ARN. Además la representación del genotipo en bits se adapta perfectamente a la representación utilizada en los algoritmos genéticos.

En la implementación de EG desarrollada en el marco de esta tesis se optó por representar el genotipo directamente como una cadena de números enteros. El principal motivo de este cambio fue un inconveniente que se observó en relación a la longitud de los genotipos tras una operación de recombinación: puede ocurrir que el operador de recombinación genere un genotipo cuya longitud no sea múltiplo de 8. En ese caso sería necesario tomar una decisión en el momento de escanear el nuevo genotipo, ya que, o bien se completa para que contenga un número de bits múltiplo de 8, o bien se trunca para conseguir el mismo efecto. Ambas opciones fueron desestimadas ya que, por una parte, la alternativa de completar el genotipo implica decidir con qué valores rellenarlo y evidentemente estos valores influyen en el proceso de búsqueda evolutiva de una manera en principio desconocida. De la misma manera, los efectos de truncar los genotipos en el proceso evolutivo tampoco parecen controlables.

Por otra parte, el uso de números enteros en lugar de bits no afecta a otros aspectos del algoritmo. Por ejemplo, el fenómeno biológico habitualmente referido como degeneración del código genético consistente en la correspondencia entre distintos codones y el mismo aminoácido sigue presente. Esto es posible porque en EG la degeneración de código genético se implementa en la operación que asocia a un codón una regla de la gramática. La figura 4.1 muestra esta circunstancia. Supóngase que una gramática contiene las reglas $<var> \rightarrow x$ y $<var> \rightarrow$ 1,0 y que el no terminal situado más a la izquierda en el fenotipo en construcción es $<var>$. Puede observarse que el resultado es el mismo con los dos tipos de representación.

Aunque el cambio en la representación del genotipo respeta el fenómeno biológico de la degeneración del código genético, por contra, condiciona a que el operador de mutación trabaje directamente sobre los codones mientras que biológicamente las mutaciones genéticas ocurren a nivel de nucleótido y no a nivel de codón. No obstante, recordando que el objetivo es reproducir algunos de los experimentos propuestos en [O'Neill y Ryan (2003)], es decir, intentar implementar el algoritmo de la manera más fiel a la propuesta por

sus creadores, es importante destacar que en EG original no se menciona en ningún momento que exista alguna correspondencia entre los nucleótidos y la secuencia de bits, y, aunque no se mencione de manera explícita, tampoco se puede deducir ya que la representación lógica sería utilizar 2 bits para un nucleótido (hay 4) y en total 6 bits para un codón, y no un total de 8 bits por codón. En el apartado 4.2 se describe la influencia del cambio de representación del genotipo en el operador de mutación.

Logicamente, también se ve afectado el operador de recombinación como se describe en el apartado 4.3.

4.2. El operador de mutación

El operador de mutación propuesto en EG original trabaja a nivel de bit. Se fija una tasa de mutación para la probabilidad de que mute un bit, y se recorre la cadena de bits, aplicando en cada uno de ellos la mutación con la tasa establecida. La manera de implementar este comportamiento consiste en recorrer la cadena de bits, y en cada uno de ellos generar un número aleatorio en el intervalo $[0, 1]$ de manera que, si el número generado es menor que la tasa prefijada, se muta el bit, y en caso contrario no se muta.

La modificación de la representación del genotipo que sustituye la cadena de bits por una cadena de números enteros implica también el cambio del operador de mutación ya que desaparecen los bits. Se establece una **tasa de mutación por genotipo**, en definitiva, una tasa de mutación por individuo. Para hacer efectiva la mutación, en los casos que proceda atendiendo a la tasa, se selecciona al azar un codón y se sustituye por un codón creado también de manera aleatoria.

Otra diferencia apreciable en el operador de mutación está relacionada con el número de codones que pueden mutar en un genotipo. En EG original, el funcionamiento del operador de mutación a nivel de bit hace posible que mute más de un codón en un genotipo mientras que, en EG ajustada, esta situación no pude ocurrir.

Para poder comparar los resultados de los experimentos que aparecen en [O'Neill y Ryan (2003)] con los resultados propios, es necesario establecer una correspondencia entre la tasa o probabilidad de mutación de un bit en EG original y la nueva tasa de mutación de un genotipo.

En EG original, la probabilidad de que mute un genotipo es 1 menos la probabilidad de que no mute, y la probabilidad de que no mute es equivalente a la probabilidad de que no mute ninguno de los bits. Si b es el número de bits de un genotipo, y p es la probabilidad de mutación de un bit, la probabilidad de que no mute el genotipo se puede calcular como la probabilidad de que no

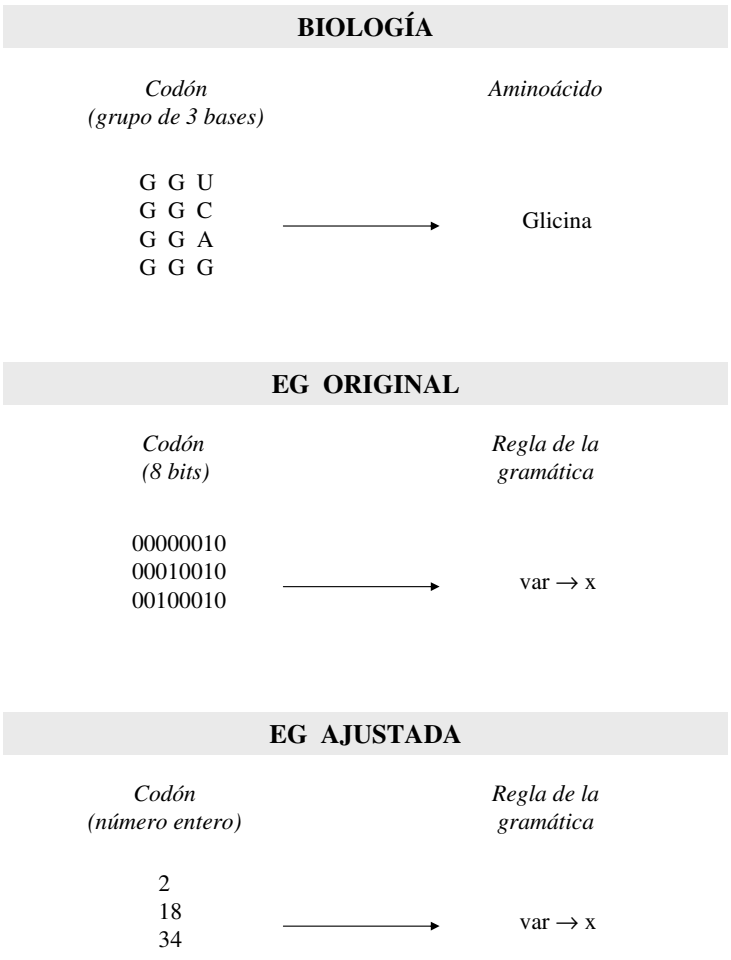

Figura 4.1: Emulación del código genético degenerado.

mute ninguno de los bits, que es $(1-p)^b$. Por lo tanto, la probabilidad de que mute un genotipo es $1-(1-p)^b$. En la tabla 4.1 se muestra la correspondencia entre el número de bits de un genotipo y la probabilidad de mutación del mismo para una tasa de mutación de bit de $0,01$, que es la propuesta en los experimentos de [O'Neill y Ryan (2003)]. Dado que en los experimentos propuestos en [O'Neill y Ryan (2003)] el tamaño inicial de los genotipos está en el rango $[1, 10]$, y que varía rápidamente con tendencia creciente a lo largo de la simulación del proceso evolutivo (ver figura 4.4), no parece inadecuado considerar un tamaño medio del genotipo de aproximadamente 30 codones, lo que correspondería a una probabilidad de mutación de genotipo de 0,91 según se puede ver en la tabla 4.1. Por este motivo existe tanta diferencia entre la tasa de mutación de los experimentos de EG original y los realizados en el contexto de esta tesis.

4.3. El operador de recombinación

El operador de recombinación propuesto en EG original es el operador de recombinación de un punto que trabaja de la siguiente manera:

1. Se seleccionan aleatoriamente dos puntos de recombinación, uno en cada genotipo.

2. Se intercambian los segmentos de la parte derecha de cada genotipo para generar dos nuevos genotipos.

En [O'Neill y Ryan (2003)] se dedica un capítulo completo a la discusión de la idoneidad del operador de recombinación de un punto, haciendo comparativas con otros métodos de recombinación, y concluyendo experimentalmente que el operador propuesto es adecuado para EG.

La representación del genotipo como una serie de números enteros implica que el intercambio de segmentos propio del operador de recombinación se realice a nivel de codón (que es el material genético que sustituye a los bits de EG original) en vez de a nivel de bit y parece difícil evaluar la repercusión de este cambio. Por ejemplo, en la figura 4.2 se muestra una operación de recombinación donde los genotipos están representados por cadenas de bits. Se puede observar que como consecuencia de que los puntos de corte están situados en puntos intermedios de codones de los genotipos de los padres, aparecen en los genotipos de los hijos secuencias de bits que dan lugar a nuevos codones que no existían en los padres. Por ejemplo, el segundo codón del hijo 1 vale 0 mientras que ninguno de los padres tiene un codón de valor 0, y lo mismo ocurre con el quinto codón del hijo 2.

número de bits del genotipo	número de codones del genotipo	probabilidad de mutación del genotipo
8	1	0,08
16	2	0,15
24	3	0,21
32	4	0,28
40	5	0,33
48	6	0,38
56	7	0,43
64	8	0,47
72	9	0,52
80	10	0,55
88	11	0,59
96	12	0,62
104	13	0,65
112	14	0,68
120	15	0,70
128	16	0,72
136	17	0,75
144	18	0,76
152	19	0,78
160	20	0,80
168	21	0,82
176	22	0,83
184	23	0,84
192	24	0,85
200	25	0,87
208	26	0,88
216	27	0,89
224	28	0,89
232	29	0,90
240	30	0,91
248	31	0,92
256	32	0,92
264	33	0,93
272	34	0,94
280	35	0,94
288	36	0,94

Tabla 4.1: Correspondencia entre el número de bits de un genotipo y la probabilidad de mutación del mismo para una tasa de mutación de bit de 0,01

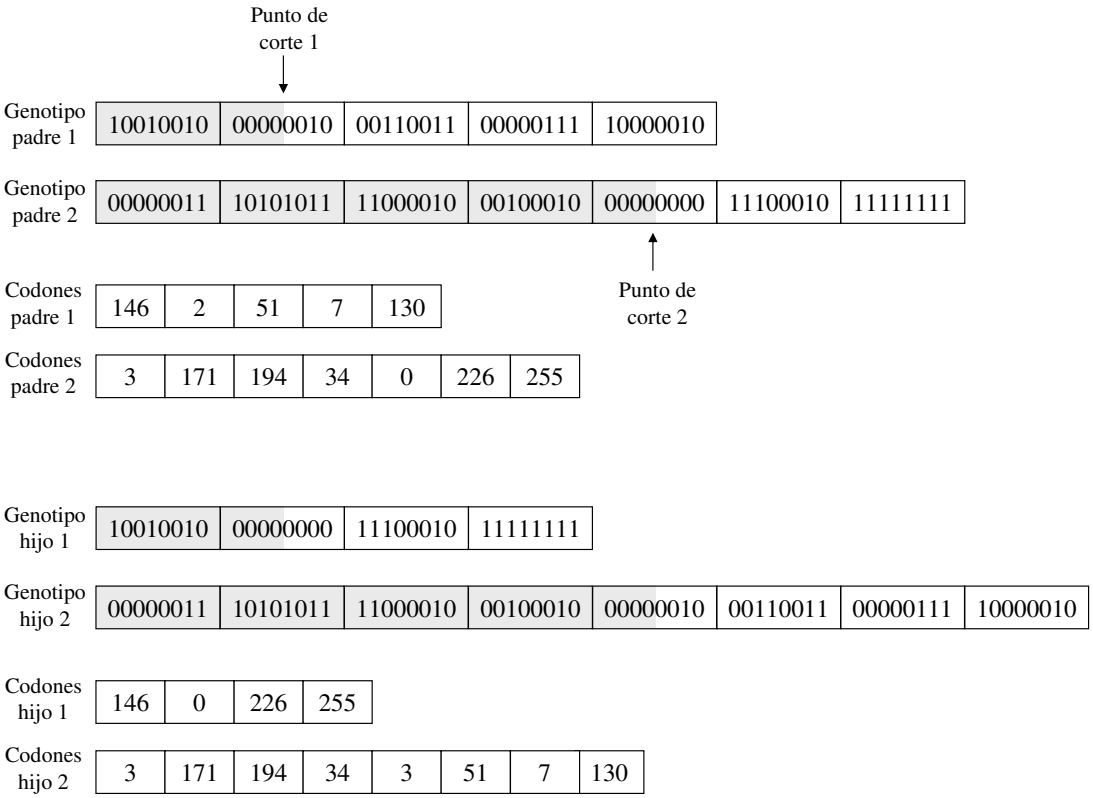

Figura 4.2: Operador de recombinación de un punto en EG.

Es evidente que la aparición de nuevos codones en los genotipos de los hijos es un suceso imposible cuando el genotipo es precisamente la secuencia de codones como puede apreciarse en el ejemplo de la figura 4.3.

No parece fácil encontrar un método que permita estimar las repercusiones directas de este fenómeno en el proceso evolutivo ya que se podría asimilar a una tasa superior de mutación. Se optó por considerar ausencia de repercusiones siempre y cuando los rendimientos de los algoritmos fueran comparables.

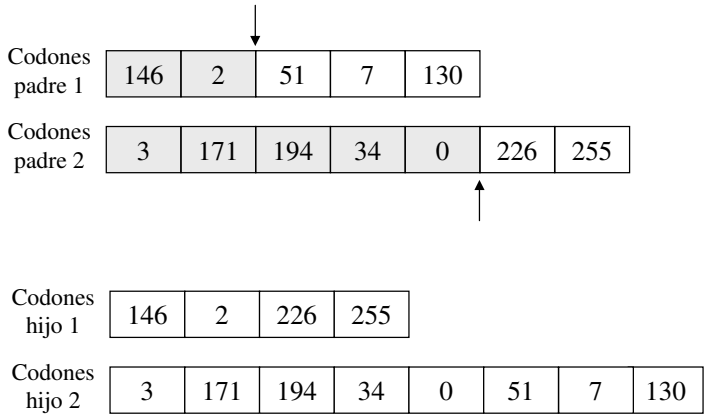

Figura 4.3: Operador de recombinación para genotipos representados mediante cadenas de números enteros.

En el proceso de intentar reproducir los experimentos propuestos en [O'Neill y Ryan (2003)] se detectó la siguiente circunstancia. El número de **codones efectivos** (los utilizados en la generación del fenotipo) es, de media, bastante inferior al número de codones que forman el genotipo. En la figura 4.4 se muestran los datos obtenidos en 500 ejecuciones del problema de regresión simbólica (en el capítulo 5 se describe en profundidad este problema). Las series representadas corresponden a las medias de los codones reales y codones efectivos en función del número de generación. Se puede observar que, a medida que crece el número de generación, el número de codones del genotipo también crece y, además, aumenta la desproporción entre número de codones reales y efectivos. Como consecuencia del fenómeno representado en la figura 4.4, a medida que avanza el proceso evolutivo, al generar de

manera aleatoria un punto de corte en el genotipo para aplicar el operador de recombinación, la probabilidad de que el punto caiga en la zona de codones no utilizados es mayor que la probabilidad de que caiga en la zona de codones efectivos, ya que ésta es menor. Por lo tanto, después de aplicar el operador de recombinación, la probabilidad de que los nuevos individuos tengan un fenotipo igual al de los padres es mayor que la probabilidad de que sean diferentes, y como consecuencia de esta circunstancia es muy habitual que el proceso de evolución se "estanque" en una solución "local" y no evolucione. Para resolver este problema, los puntos de corte en los genotipos se generaron utilizando como cota superior el número de codones efectivos en lugar del número de codones reales. Así, se observó la reducción drástica del proceso de "estancamiento" descrito anteriormente.

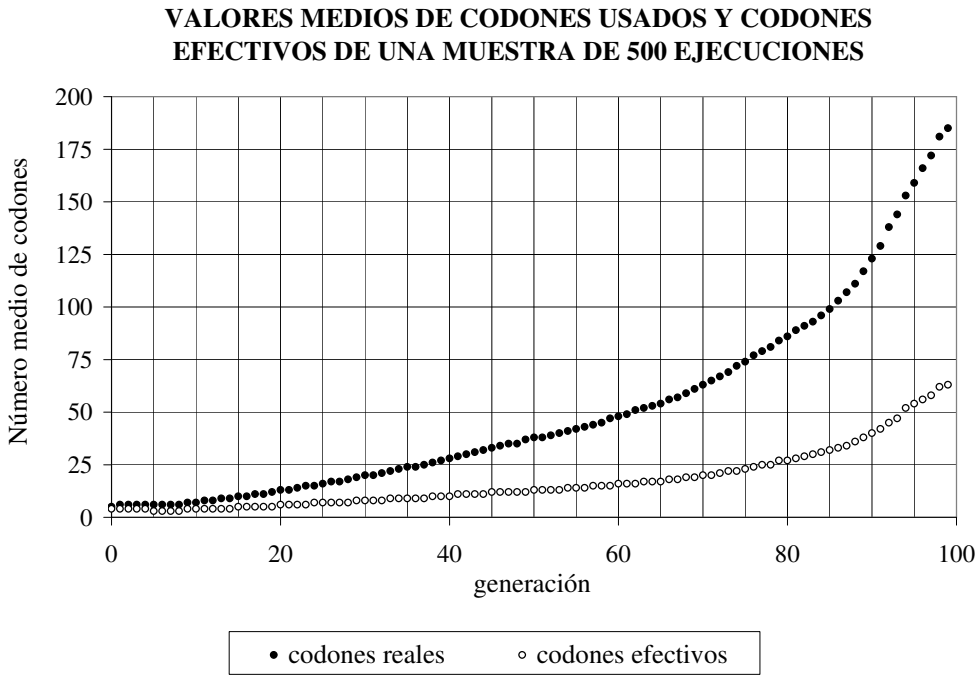

Figura 4.4: Comparativa de número medio de codones efectivos y número medio de codones reales.

El operador de reducción descrito en [Ryan y otros (1998)] parece que tiene el mismo objetivo que el mecanismo propuesto anteriormente. Dicho

operador se aplica con una determinada probabilidad a los individuos que no expresan todos sus genes y funciona eliminando el material genético que no se utiliza en el proceso de traducción de genotipo a fenotipo.

4.4. Otros operadores

En EG original se proponen dos operadores más junto con el de mutación y el de recombinación, que son el operador de *wrapping* y el operador de duplicación (ambos descritos en el capítulo 2)

El operador de wrapping sí se ha implementado en la herramienta de software desarrollada como parte de este trabajo, pero no se ha utilizado de manera habitual en los experimentos. No obstante, este operador es el menos afectado por el cambio de representación del genotipo ya que el resultado de aplicar el operador es similar, ya que su funcionamiento implica únicamente la reutilización de código genético de un individuo.

El operador de duplicación no se ha incorporado en la versión ajustada de EG y se reserva junto con otros operadores inspirados en la naturaleza para futuras líneas de trabajo.

4.5. El algoritmo evolutivo

En cuanto al algoritmo evolutivo utilizado en la versión ajustada de EG solamente se modificó la política de reemplazo de la población sustituyendo la técnica de estado estacionario (*steady state*) propuesta en [O'Neill y Ryan (2003)] por el reemplazo generacional con elitismo opcional. El uso de elitismo significa que de una generación a otra se mantiene el mejor individuo (para conservar su información genética).

La motivación del cambio de política de reemplazo fue la obtención de rendimientos muy bajos comparativamente con los mostrados en [O'Neill y Ryan (2003)] cuando se utilizaba la técnica de estado estacionario. La medida del rendimiento del algoritmo fue un punto problemático durante el desarrollo de los experimentos para la puesta a punto de la EG ajustada como se describe detalladamente en el apartado 4.6.

4.6. Evaluación del rendimiento del algoritmo

4.6.1. Antecedentes

La implementación del método de EG ajustada como parte de la herramienta **GEEMMA** desarrollada en el contexto de esta tesis fue seguida de la fase de pruebas que comenzó con el primer ejemplo propuesto en [O'Neill y Ryan (2003)], que aborda el problema de la regresión simbólica (ver capítulo 5 para una descripción detallada). El hecho de encontrar rendimientos mucho perores que los presentados en [O'Neill y Ryan (2003)] motivó la realización de numerosas pruebas, como por ejemplo:

- variación de número de bits de cada codón
- variación del número de codones del genotipo
- variación en el número máximo de generaciones
- variación en el tamaño de la población
- variación de las tasas de aplicación de los operadores de mutación y recombinación
- variación de la configuración del operador de *wrapping* barriendo el espectro que va desde no aplicar *wrapping* hasta la aplicación de 1, 2, 3, ... veces.
- sustitución de la estrategia de reemplazo de estado estacionario por la técnica de reemplazo generacional con elitismo opcional

Después de la realización de las pruebas, solamente se obtuvo una mejora del rendimiento con el cambio de estrategia de reemplazo, pero no se llegó nunca a obtener los resultados mostrados en [O'Neill y Ryan (2003)]. Para el problema de regresión simbólica se obtenía una probabilidad acumulada de éxito del orden del 5 % para la generación 25 frente al 85 % esperado, y en la generación 50 se alcanzaba solamente una probabilidad acumulada de éxito del orden de 60-70 % frente al prácticamente 100 % esperado. En la figura 4.5 puede observarse la disparidad entre las curvas.

La realización de tantas pruebas supuso una inversión muy importante de tiempo y esfuerzo (en primera instancia infructuoso). En ningún momento se pensó que los valores de las gráficas que aparecen en [O'Neill y Ryan (2003)] pudieran estar alterados por algún error en su edición o algo similar

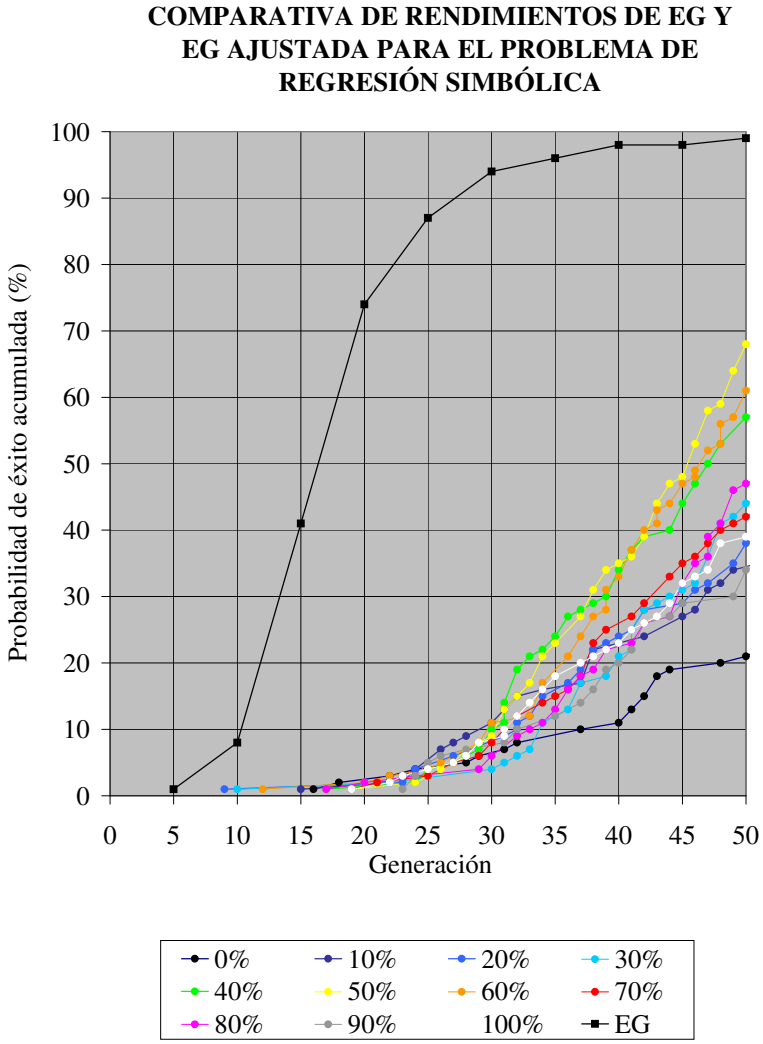

Figura 4.5: Comparativa de rendimientos para el problema de regresión simbólica resuelto con EG y EG ajustada.

ya que, iban acompañadas de su correspondiente explicación con referencias a los valores que aparecen en las mismas. En ocasiones, es posible detectar incorrecciones en los valores que aparecen en gráficas correspondientes a experimentos científicos, gracias a la explicación que habitualmente se adjunta. Este no era el caso ya que coincidían los valores de las gráficas con sus correspondientes explicaciones.

El hecho de que las gráficas de rendimiento de EG para el problema de regresión simbólica contuviesen también, a modo de comparación, los resultados obtenidos con la técnica de programación genética (PG) de Koza [Koza (1992)] motivó acudir a la fuente para cotejar los valores que mostraban las gráficas. En la figura 4.6 se muestran las dos series de rendimiento de la técnica de programación genética para el problema de regresión simbólica obtenidas de las dos referencias bibliográficas citadas previamente, el libro de los autores de EG [O'Neill y Ryan (2003)], y uno de los libros de Koza [Koza (1992)].

La disparidad evidente de los rendimientos que aparecen en la figura 4.6 y la lectura detallada de las explicaciones de Koza relativas a la evaluación del rendimiento podrían sugerir que los valores que aparecen en las gráficas de rendimiento en [O'Neill y Ryan (2003)] pudieran no corresponder a la frecuencia acumulada de éxito sino a otra magnitud derivada de la misma. En la sección 4.6.2 se resume la propuesta de Koza en [Koza (1992)] para evaluar rendimientos.

4.6.2. Propuesta de Koza para evaluar el rendimiento

Koza propone un método para medir el rendimiento del modelo de programación genética en términos del coste computacional necesario para resolver un problema [Koza (1992)]. En concreto la medida propuesta es el **número de individuos que se deben procesar para resolver el problema con una cierta probabilidad**. Dicha medida proporciona una estimación de la dificultad del problema.

En primer lugar se estima de manera experimental la **probabilidad instantánea** $Y(M, i)$ de que en una ejecución independiente del experimento sobre una población de tamaño M se obtenga una solución en la generación i. La estimación de $Y(M, i)$ requiere un elevado número de experimentos (y su fiabilidad crece con el número de experimentos)

A partir de la probabilidad instantánea $Y(M, i)$ estimada para los distintos valores de i se calcula la **probabilidad acumulada de éxito** $P(M, i)$.

La **probabilidad de encontrar al menos una solución antes de la generación** i **en** R **ejecuciones independientes** es $1 - [1 - P(M, i)]^R$. Es precisamente esta magnitud la que podría estar representada, en lugar

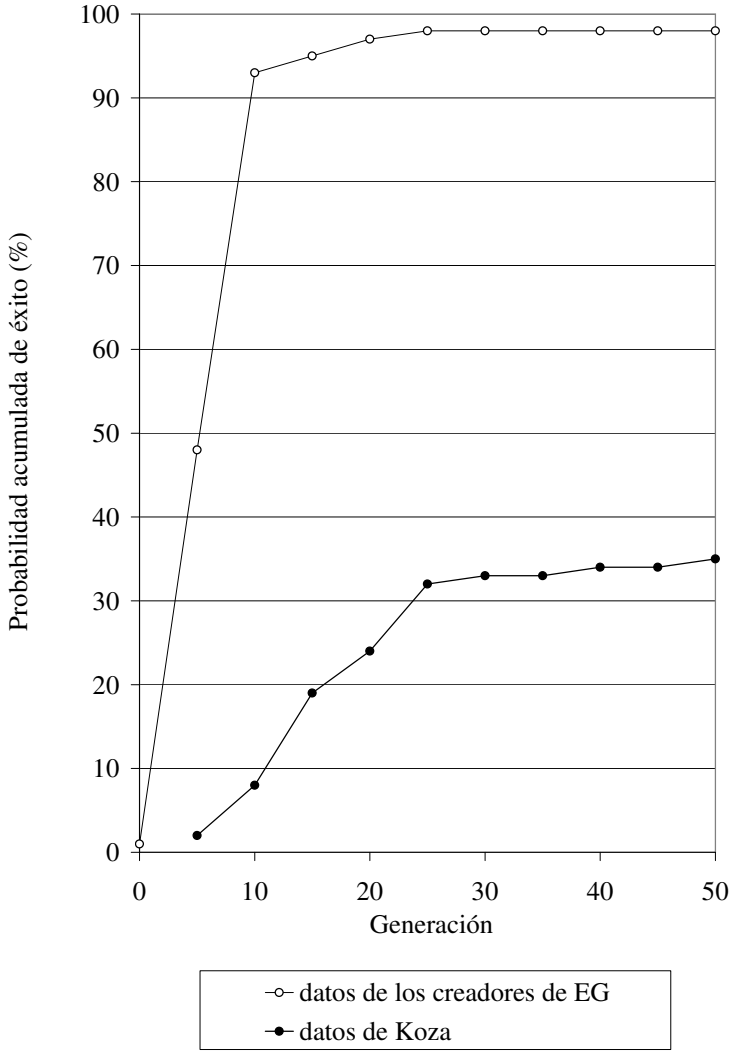

Figura 4.6: Rendimiento de la técnica de programación genética para el problema de regresión simbólica.

de la probabilidad acumulada de éxito, en las gráficas de rendimiento del problema de regresión simbólica que aparecen en [O'Neill y Ryan (2003)]. En la figura 4.7 se muestran de nuevo las dos series de rendimiento ya presentadas en la figura 4.6, y con trazo de color, se representan las series correspondientes a la magnitud $1 - [1 - P(M, i)]^R$ para distintos valores de R. La semejanza entre éstas últimas series y los valores de rendimiento que aparecen en el libro de EG motivó la hipótesis relativa a la magnitud representada en el eje de ordenadas.

En cualquier caso, en el contexto del presente trabajo, el rendimiento de los métodos se evaluará generalmente en términos de probabilidad acumulada de éxito. La discusión previa simplemente pretendía describir la hipótesis que se planteó para explicar la gran diferencia entre el rendimiento descrito por los autores de EG para el problema de regresión simbólica y el rendimiento obtenido con la herramienta desarrollada como parte de esta tesis.

No obstante, esta sección continúa con un resumen del planteamiento de Koza para la evaluación del rendimiento de los métodos.

Si se quiere encontrar una solución con una probabilidad $z = 1 - \varepsilon = 99\,\%$, entonces se debe cumplir que:

$$z = 1 - [1 - P(M, i)]^R$$

El número de ejecuciones independientes $R(z)$ para encontrar una solución antes de la generación i con una probabilidad $z = 1 - \varepsilon = 99\,\%$ depende de z y de $P(M, i)$.

Tomando logaritmos, se obtiene que:

$$R(z) = \left\lceil \frac{log(1 - z)}{log(1 - P(M, i))} \right\rceil = \left\lceil \frac{log\varepsilon}{log(1 - P(M, i))} \right\rceil \tag{4.1}$$

donde $\varepsilon = 1 - z = 0,01$ y los símbolos $\lceil$ y $\rceil$ representan a la función de redondeo al entero superior más próximo.

En la figura 4.8 se muestra el gráfico del número de ejecuciones independientes necesarias $R(z)$ en función de la probabilidad acumulada de éxito $P(M, i)$ para $z = 99\,\%$. Por ejemplo, si la frecuencia acumulada de éxito $P(M, i)$ es $0,09$ son necesarias 48 ejecuciones independientes para encontrar una solución con una probabilidad del $99\,\%$. Si $P(M, i)$ es $0,68$ sólo son necesarias 4 ejecuciones independientes; si $P(M, i)$ es $0,78$ sólo son necesarias 3 ejecuciones independientes; y si $P(M, i)$ es $0,90$ sólo son necesarias 2 ejecuciones. Estos tres valores de $P(M, i)$, $68\,\%$, $78\,\%$ y $90\,\%$ tienen una especial relevancia ya que son los porcentajes más pequeños para los cuales sólo se necesitan 4, 3 y 2 ejecuciones independientes respectivamente, para encontrar

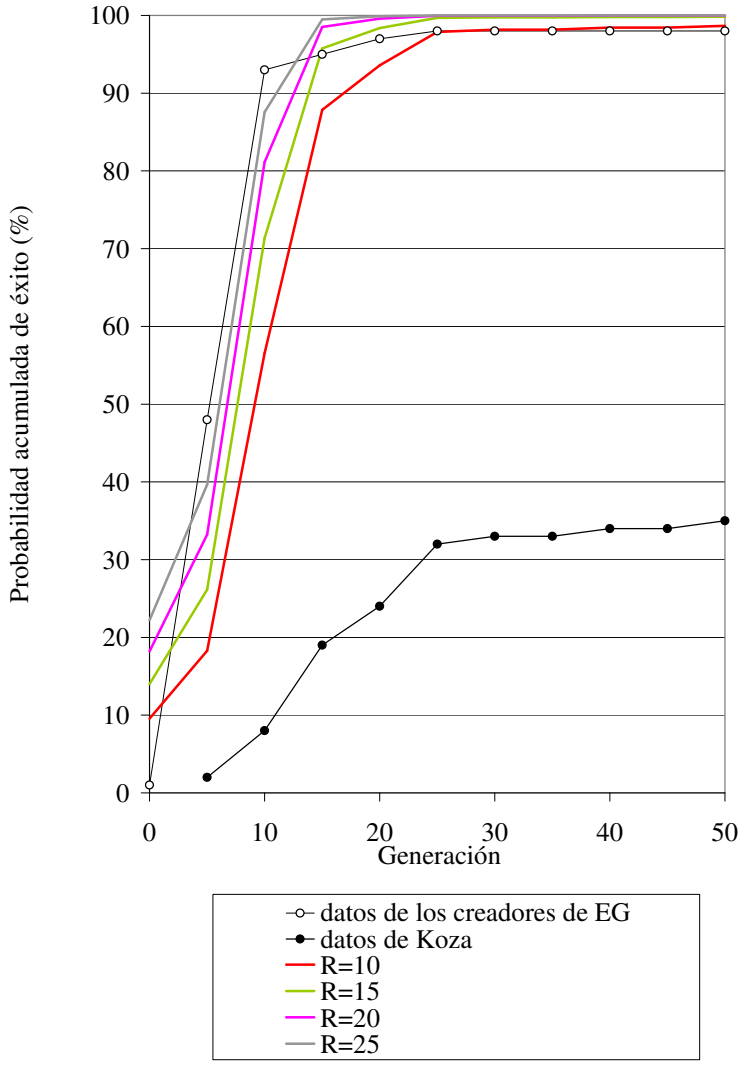

Figura 4.7: Rendimiento de la técnica de programación genética para el problema de regresión simbólica.

una solución con una probabilidad $z = 99\,\%$. En el caso extremo en el que $P(M, i)$ es $0,99$ sólo es necesaria una ejecución.

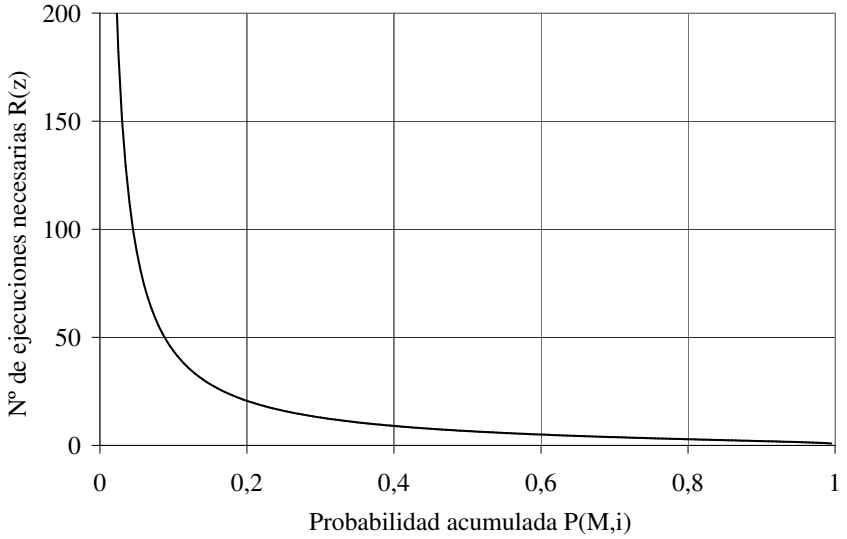

Figura 4.8: Número de ejecuciones independientes necesarias R(z) en función de la probabilidad acumulada de éxito P(M,i) para z=99 %

Para estimar el coste computacional para resolver un problema con una cierta probabilidad z, con un tamaño de población M y en un número máximo de generaciones i en términos del número de individuos procesados, $I(M, i, z)$ bastaría con obtener el número de ejecuciones requeridas $R(z)$ para la frecuencia acumulada $P(M, i)$ y multiplicarlo por el número de generaciones i y por el tamaño de la población M. El valor obtenido de esta manera para $I(M, i, z)$ está sobrestimado ya que puede ocurrir que se encuentre más de una solución, o bien que se encuentre antes de la generación i.

El tamaño de la población M y el número máximo de generaciones G son dos parámetros fundamentales en la configuración de un algoritmo genético.

Para un tamaño de población determinado M la frecuencia acumulada $P(M, i)$ incrementa si se continúa el experimento durante más generaciones. En principio, el operador de mutación permite que se alcance cualquier punto del espacio de soluciones si el experimento se ejecuta durante un número alto de generaciones. No obstante, existe un punto a partir del cual el coste de

prolongar la ejecución del experimento durante más generaciones es superior al beneficio obtenido por el incremento de $P(M, i)$.

Para el problema de regresión simbólica, que se describe completamente en el capítulo 5 se ha obtenido experimentalmente la curva de probabilidad acumulada de éxito realizando 1000 ejecuciones independientes con una población $M = 500$ y un número máximo de generaciones $G = 150$. Los resultados se muestran en la figura 4.9.

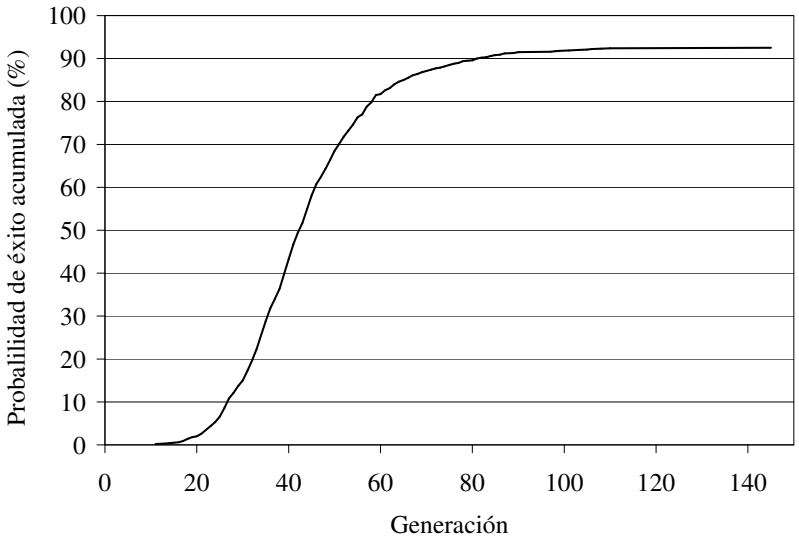

Figura 4.9: Probabilidad acumulada de éxito $P(M, i)$ para el problema de regresión simbólica

En la tabla 4.2 se muestra el número total de individuos que se deben procesar para encontrar una solución con una probabilidad $z = 99\,\%$ para distintas generaciones. Se puede observar que existe un punto, en concreto el correspondiente a la generación 82, en el que se alcanza un mínimo en el número de individuos que se deben procesar, a partir del cual el coste de prolongar la ejecución del experimento durante más generaciones es superior al beneficio obtenido por el incremento de $P(M, i)$.

La probabilidad acumulada de éxito $P(M, i)$ calculada experimentalmente para la generación 20 es del $2\,\%$. Con la ecuación (4.1) se obtiene que el número de ejecuciones necesarias para obtener una solución con una probabilidad del $99\,\%$ antes de la generación 20 es $R(z) = 228$. Para calcular

Número de generación i	Probabilidad acumulada de éxito $P(M,i)$	Número de ejecuciones necesarias $R(z)$	Número total de individuos procesados $I(M,i,z)$
20	2 %	228	2.280.000
30	15 %	29	435.000
40	43,3 %	9	180.000
60	81,7 %	3	90.000
82	90,2 %	2	82.000
103	92 %	2	103.000
145	92,5 %	2	145.000

Tabla 4.2: Número de individuos que deben ser procesados para encontrar una solución con un 99 % de probabilidad para el problema de regresión simbólica.

el número de individuos que se necesita procesar bastaría con realizar el producto $500 \times 228 \times 20 = 2{,}280{,}000$.

Los cálculos realizados para la generación 30, que tiene una probabilidad acumulada de éxito $P(M,i)$ del 15 % permiten afirmar que es necesario procesar $500 \times 29 \times 30 = 435{,}000$ individuos para obtener una solución con una probabilidad de 99 %.

Para las generaciones 40 y 60 con probabilidades acumuladas de éxito de 43,3 % y 81,7 % respectivamente, el número de ejecuciones necesarias para obtener una solución con una probabilidad del 99 % es, en cada caso, 9 y 3, de donde se deduce el número de individuos que se necesita procesar para la generación 40 es $500 \times 9 \times 40 = 180{,}000$, y para la generación 60 se reduce a $500 \times 3 \times 60 = 90{,}000$.

El descenso brusco que se observa en el número de individuos que se deben procesar desde la generación 20 hasta la 60 es debido al rápido ascenso de la probabilidad acumulada.

Como puede observarse en la tabla 4.2 en la generación 82 el número de individuos que se necesita procesar alcanza el mínimo, 82.000 individuos, y en generaciones posteriores crece el valor, en concreto para la generación 103 se necesita procesar 103.000 individuos y en la 145 asciende a 145.000.

En la figura 4.10 se muestra, para el problema de regresión simbóli-

ca, la curva de probabilidad acumulada de éxito $P(M, i)$ obtenida experimentalmente a partir de 1000 ejecuciones independientes con una población $M = 500$ y un número máximo de generaciones $G = 150$, junto con la curva $I(M, i, z)$de número de individuos que se necesita procesar para obtener una solución con una probabilidad $z = 99\,\%$.

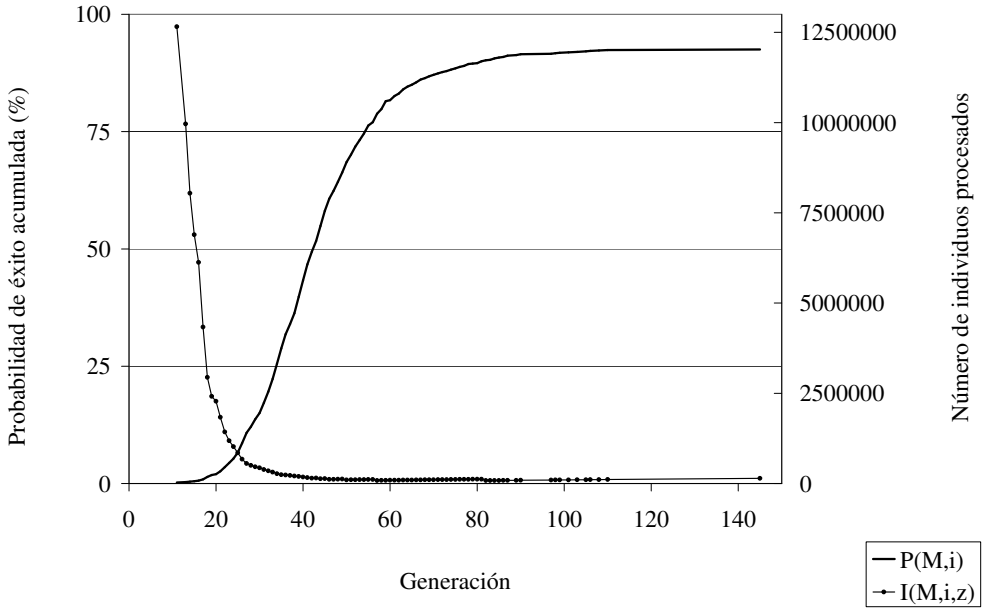

Figura 4.10: Probabilidad acumulada de éxito $P(M, i)$ y número de individuos que se necesita procesar $I(M, i, z)$ para el problema de regresión simbólica.

En la figura 4.10 la amplitud de la escala no permite ver detalladamente la forma de la función $I(M, i, z)$ que sí se puede observar en la figura 4.11. En primer lugar, se ve claramente que el mínimo se encuentra en la generación 82. La forma de dientes de sierra se explica de la siguiente manera. Los tramos lineales crecientes corresponden a generaciones para las cuales el número de ejecuciones independientes calculado con la ecuación (4.1) es el mismo, y por lo tanto, el número de individuos $I(M, i, z)$ que se obtiene como el producto de la generación, el tamaño de la población y el número de ejecuciones independientes, al ser constante el último factor, crece proporcionalmente al tamaño de la población.

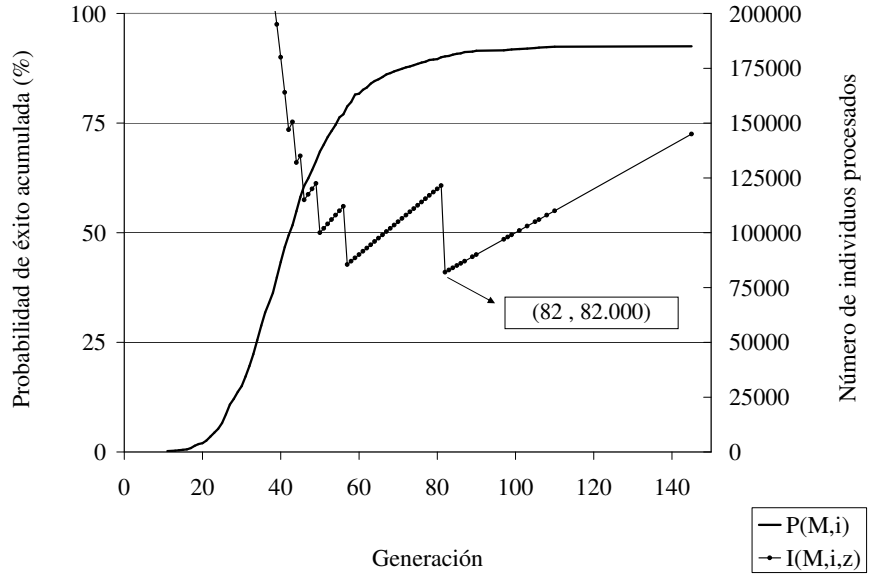

Figura 4.11: Probabilidad acumulada de éxito $P(M, i)$ y número de individuos que se necesita procesar $I(M, i, z)$ para el problema de regresión simbólica.

El número de individuos que se procesan es una medida de la dificultad del problema para un determinado tamaño de la población. El estudio realizado hasta el momento permite deducir el número de generación para el cual es mínimo el número de individuos que se procesan para un tamaño de población prefijado. También se puede estudiar el tamaño de población óptimo de una manera similar.

Se han realizado pruebas para el problema de regresión simbólica y tamaños de población de 1000, 2000 y 4000 individuos. En la figura 4.12 se muestran los datos obtenidos de probabilidad acumulada de éxito $P(M, i)$ y de número de individuos que se necesita procesar $I(M, i, z)$ para un tamaño de población $M = 1000$. El mínimo de la función $I(M, i, z)$ se alcanza en

la generación 45 de manera que sería necesario procesar 135.000 individuos para encontrar una solución al problema con una probabilidad del 99 %.

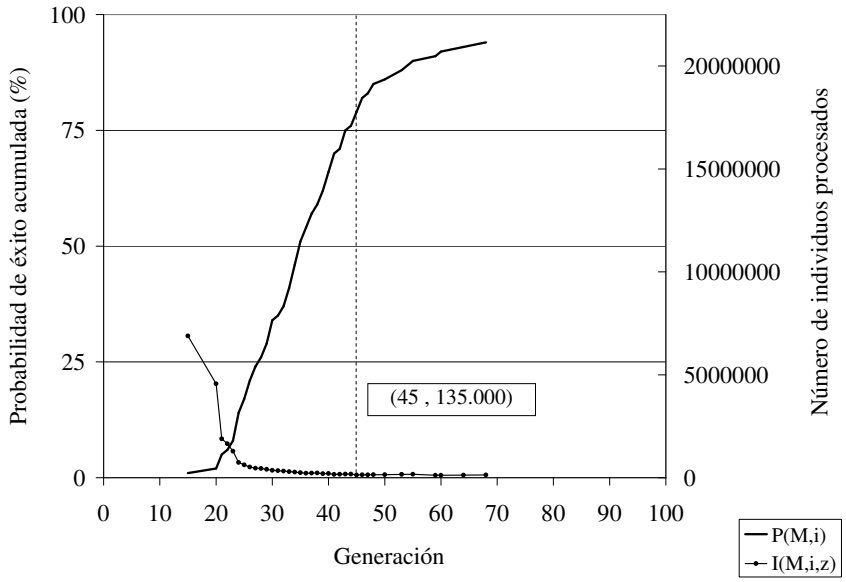

Figura 4.12: Probabilidad acumulada de éxito $P(M, i)$ y número de individuos que se necesita procesar $I(M, i, z)$ para el problema de regresión simbólica con un tamaño de población $M = 1000$.

Si se incrementa el tamaño de la población a 2000 individuos, se obtienen las curvas de rendimiento que se muestran en la figura 4.13. En este caso, disminuye a 100.000 el número de individuos que se necesita procesar para obtener una solución con una probabilidad del 99 %, correspondiendo este mínimo a la generación 50 que tiene una probabilidad acumulada de éxito del 100 % y por lo tanto el número de ejecuciones independientes $R(z)$ desciende a 1, y el número de individuos que se deben procesar pasa a ser $2000 \times 1 \times 50 = 100.000$. Este valor es menor que el valor obtenido para un tamaño de población de 1000 que requería procesar 135.000 individuos.

Por último, aumentando el tamaño de la población a 4000, no se mejora el rendimiento ya que el mínimo de la función $I(M, i, z)$ es 208.000 en la generación 52. En la figura 4.14 se muestran los resultados.

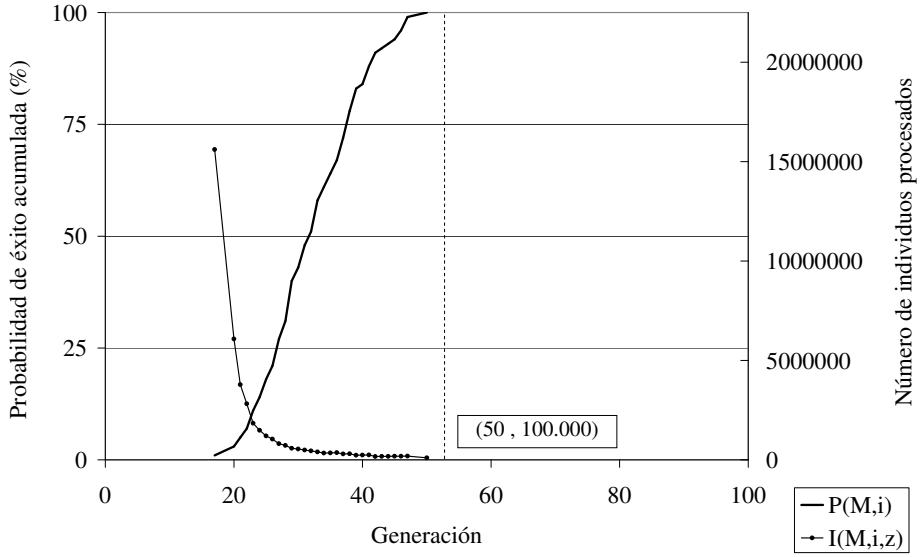

Figura 4.13: Probabilidad acumulada de éxito $P(M, i)$ y número de individuos que se necesita procesar $I(M, i, z)$ para el problema de regresión simbólica con un tamaño de población $M = 2000$.

En resumen, los rendimientos obtenidos para tamaños de población 1000, 2000 y 4000, utilizando como medida del rendimiento el número de individuos que se deben procesar para obtener una solución al problema con una probabilidad del 99 %, que son 135.000, 100.000 y 208.000 individuos respectivamente, no mejoran el rendimiento obtenido para una población de 500 individuos en cuyo caso el rendimiento es de 82.000 individuos.

En definitiva, el esfuerzo computacional para resolver un problema medido como número de individuos que se necesitan procesar permite evaluar su dificultad y como consecuencia proporciona un mecanismo para

comparar la dificultad de resolver diferentes problemas.

Además, para la resolución de nuevos problemas puede ser interesante disponer de curvas de rendimiento de problemas que se consideran de dificultad similar, ya que, se pueden utilizar para estimar el tamaño óptimo de la población así como el número máximo de generaciones que se debe utilizar para obtener el mayor rendimiento.

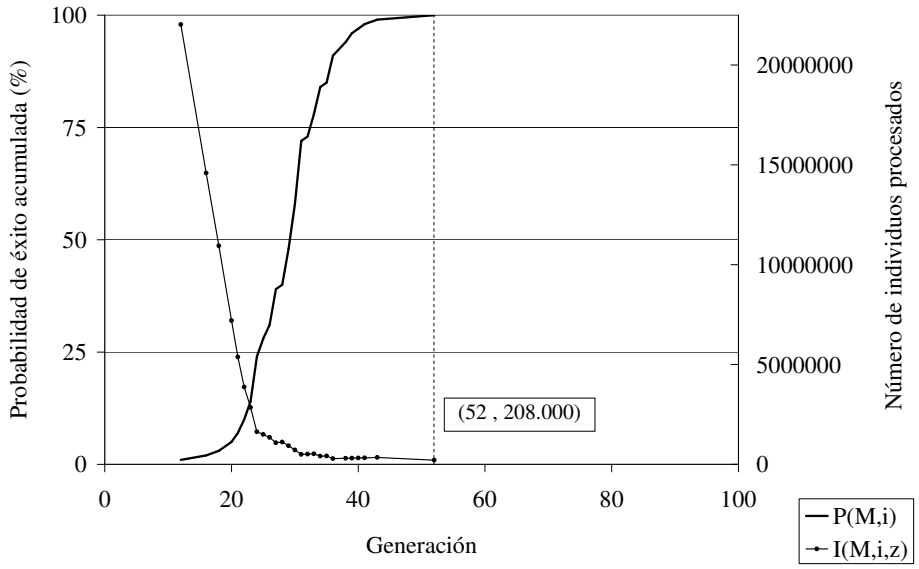

Figura 4.14: Probabilidad acumulada de éxito $P(M, i)$ y número de individuos que se necesita procesar $I(M, i, z)$ para el problema de regresión simbólica con un tamaño de población $M = 4000$.

Capítulo 5

Evolución con gramáticas de atributos

Los autores de EG plantean en [O'Neill y Ryan (2003)] la incorporación de contenido semántico a su modelo como una futura línea de investigación. En el capítulo 2 se puede leer una revisión de las distintas propuestas en esa línea que se han encontrado durante la fase de estudio del estado del arte del área en el que se enmarca este trabajo.

La primera propuesta de esta tesis es incorporar semántica a EG haciendo uso de gramáticas de atributos como ejemplo de *especificación estática* de contenido semántico.

En este capítulo se describe en profundidad esta primera propuesta y se muestran los resultados de su aplicación en algunos ejemplos clásicos del área como por ejemplo los problemas de regresión e integración simbólica. Se utilizan los dos modelos, por un lado con EG y por otro con la nueva propuesta de evolución gramatical con semántica expresada con gramáticas de atributos, para así, poder realizar un estudio comparativo de los resultados, y en el mejor de los casos extraer conclusiones acerca del nuevo modelo.

5.1. Objetivo

El objetivo de incorporar semántica a EG es proporcionar un mecanismo que permita durante el proceso evolutivo la generación de individuos sintáctica y semánticamente correctos (que satisfagan un conjunto de restricciones semánticas previamente establecidas) Obviamente, las restricciones semánticas son dependientes del problema concreto que se pretende resolver. De esta manera, la población estará formada por individuos correctos de acuerdo con la semántica establecida, y en principio, se puede suponer que la "calidad" de

la población mejorará y que, por tanto, aumentará el rendimiento del proceso de búsqueda. Parece razonable pensar que en los problemas en los que el espacio de búsqueda es muy extenso, puede resultar beneficioso incrementar la calidad de los individuos. Se entiende que un individuo es de mayor calidad si se ajusta a las restricciones semánticas previamente establecidas. Por lo tanto, la definición de las restricciones semánticas es el aspecto más relevante en el planteamiento de la solución de problemas mediante evolución gramatical con semántica expresada mediante gramáticas de atributos (a partir de ahora EGA)

5.2. El método

El método propuesto en esta tesis consiste en sustituir la gramática independiente del contexto del modelo de EG por una gramática de atributos [Knuth (1968) y Knuth (1971)] con el objetivo de generar individuos semánticamente correctos de acuerdo con la semántica expresada mediante la gramática de atributos. Una gramática de atributos permite definir, para cada símbolo, uno o más atributos y, para cada regla, un conjunto de acciones semánticas que tienen como objetivo calcular los valores de los atributos de los símbolos de la regla. Según se describe en [Alfonseca y otros (2007)] los atributos se pueden clasificar en dos grupos: sintetizados y heredados. Los atributos sintetizados pertenecen a los símbolos no terminales que aparecen en la parte izquierda de las reglas. Sus valores se calculan a partir de los valores de los atributos de los símbolos de la parte derecha de la regla. Los atributos heredados son los de los símbolos terminales y no terminales de la parte derecha de las reglas y cuyos valores se calculan utilizando los atributos de la parte izquierda o los de otros símbolos de la parte derecha. Se pueden distinguir dos tipos de atributos heredados. Por una parte, los atributos heredados por la derecha cuando el cálculo del valor de un atributo utiliza atributos de los símbolos que están situados a su derecha. Por otra parte, los atributos heredados por la izquierda cuando sólo se utilizan los que están a su izquierda, ya sea en la parte derecha de la regla o en la parte izquierda.

El método propuesto no sería completo si además de proponer la sustitución de la gramática independiente del contexto por una gramática de atributos no redefiniera el algoritmo de transformación de genotipo a fenotipo, que establece la secuencia de ejecución de las acciones asociadas a las reglas mediante las cuales se calculan los valores de los atributos tanto sintetizados como heredados. Por lo tanto, el método también incluye la elección de la secuencia o instantes de ejecución de las acciones semánticas. Se ha

utilizado para ello como referencia las técnicas utilizadas en el desarrollo de compiladores [Aho y otros (2008), Grune y otros (2007), Alfonseca y otros (2007) y Levine y otros (1992)] en concreto en las tareas de análisis semántico y generación de código. En los compiladores que realizan análisis sintáctico descendente, los atributos que se adaptan de manera natural son los atributos heredados y su cálculo o evaluación se puede realizar fácilmente en el momento en que se utiliza una regla para expandir un nodo del árbol de análisis. Por el contrario, en los compiladores basados en análisis sintáctico ascendente los atributos más adecuados son los sintetizados y es posible realizar su evaluación cada vez que se reduzca una regla de la gramática en el proceso de reducción del axioma. Lógicamente, en EGA, las acciones semánticas asociadas a las reglas se ejecutan durante el proceso de traducción de genotipo a fenotipo, y el método/modelo define exactamente cuándo y cómo, según queda descrito en el apartado 5.2.1.

En síntesis, el primer método/modelo desarrollado en el contexto de este trabajo:

1. remplaza la gramática independiente del contexto del modelo de EG por una gramática de atributos.

2. redefine el algoritmo de transformación de genotipo a fenotipo estableciendo la secuencia de ejecución de las acciones semánticas de la gramática de atributos.

5.2.1. Algoritmo de generación del genotipo desde el fenotipo en EGA

El algoritmo de generación del fenotipo desde el genotipo de EGA propuesto en este trabajo añade al algoritmo de generación de EG la construcción explícita del árbol de generación, al que se añade en cada nodo, el valor de los atributos del símbolo del nodo a partir de la gramática de atributos que sustituye a la gramática independiente del contexto de EG. Los tipos de atributos evaluables en el algoritmo son los siguientes:

- Atributos heredados por los nodos hijos de su nodo padre (heredados por la izquierda)

- Atributos heredados de los hermanos situados a la izquierda de un nodo dado (heredados por la izquierda)

- Atributos sintetizados, que corresponden a los atributos de los padres calculados a partir de los de los hijos.

Puede observarse que no se utilizan atributos heredados por la derecha. La justificación de este desuso se pospone a la descripción del algoritmo.

Puede resumirse brevemente el proceso de traducción de genotipo a fenotipo de EG: se parte del axioma y se construye una derivación por la izquierda. Para ello se utiliza la información del genotipo para seleccionar la regla de la gramática que se aplica para expandir el no terminal en curso (siempre el situado más a la izquierda del fenotipo en desarrollo) La construcción del árbol de derivación es similar. Se inicia construyendo el nodo raíz con el axioma de la gramática, y a partir de ese momento el proceso consiste en la expansión del nodo pendiente de expandir situado más a la izquierda y en el nivel más profundo del árbol con retroseguimiento. Se observan las siguientes reglas:

1. Cuando se expande un nodo se realizan las siguientes tareas:

   a) Se crean los nodos de los símbolos de la parte derecha de la regla y se asignan como hijos al nodo expandido manteniendo su orden.

   b) Si hay atributos heredados por los hijos (símbolos de la parte derecha) del padre (símbolo de la parte izquierda), se calculan en este instante

2. Se considera que termina el procesamiento de un nodo cuando ha terminado el procesamiento de todos sus hijos. Teniendo en cuenta que los nodos hoja del árbol correspondientes a símbolos terminales no tienen hijos, cuando se crea un nodo hoja también se termina su procesamiento. Las tareas vinculadas a la finalización del procesamiento de los nodos son las siguientes:

   a) Si el nodo cuyo procesamiento ha terminado tiene hermanos, se comprueba si hay atributos heredados entre hermanos y todos aquellos que se puedan evaluar se calculan en este momento.

   b) Cuando termina el procesamiento de todos los nodos hijos de un nodo padre, se evalúa los atributos sintetizados del padre a partir de los valores de los atributos de los hijos.

3. El axioma de la gramática corresponde a la raíz del árbol y por lo tanto no tiene padre ni hermanos y en consecuencia no puede tener atributos heredados, únicamente atributos sintetizados. Aunque existe un uso especial de atributos heredados por la raíz del árbol, que sirve para proporcionar "entradas´´ al árbol. Cuando se termina el proceso del

nodo raíz se ha terminado con la construcción del árbol de derivación y con la evaluación de los atributos semánticos de todos sus nodos.

4. Los nodos hoja no tienen hijos y por lo tanto no pueden tener atributos sintetizados. Otro mecanismo para proporcionar entradas al árbol, similar al mencionado anteriormente para la raíz, consiste en asignar valores (sintetizados) a estos nodos.

5. En las operaciones de expansión y terminación de un nodo es posible consultar los valores de los atributos de los símbolos de la producción implicada en la operación ya que dichos valores permiten establecer la corrección semántica de un individuo. Si durante la ejecución del algoritmo de traducción de genotipo a fenotipo se detecta la violación de alguna de las condiciones establecidas sobre los atributos, se interrumpe el proceso de traducción y se rechaza el individuo cuyo fenotipo está siendo generado.

La manera en la que se construye el árbol de derivación, en profundidad por la izquierda con retroseguimiento, determina el desuso de atributos heredados por la derecha. Efectivamente, si un nodo tuviera atributos heredados por la derecha, éstos se evaluarían una vez que el propio nodo hubiera terminado, y por lo tanto, serían inservibles a efectos de su uso en el subárbol que hipotéticamente colgara del propio nodo.

Para aclarar el funcionamiento del algoritmo se muestra un ejemplo en el que el fenotipo de los individuos de la población está formado por un número natural mayor que cero seguido de tantas tramas de dígitos binarios como indique dicho número. Por otra parte, no existe restricción en cuanto al número máximo de tramas ni a la longitud de las mismas. De acuerdo con la descripción anterior, los siguientes fenotipos serían semánticamente correctos ya que en ambos el número natural que aparece al principio de la cadena coincide con el número de tramas binarias:

```
3    000    11    10
4    1111    1100    1010101010101    0
```

Y serían semánticamente incorrectos los siguientes fenotipos:

```
0    000    11    10
8    1010    100
2    1111111    1    0000000    1010101
```

el primero porque el número natural inicial no es mayor que 0, el segundo porque el número de tramas es inferior al que debería ser, y el tercero porque

el número de tramas supera al indicado al comienzo de la cadena.

La relación entre el número natural y la cantidad de tramas es una restricción semántica que no se puede expresar con una gramática independiente del contexto pero sí con una gramática de atributos. Para simplificar la exposición, en primer lugar definimos una gramática independiente del contexto G que genere cadenas formadas por un número natural seguido de una secuencia de tramas sin ninguna relación entre el número y la cantidad de tramas. Posteriormente, añadiremos a G un sistema de atributos para incorporar las restricciones semánticas del ejemplo y quedar así definida la gramática de atributos G_A. Vamos a usar G_A como ejemplo para clarificar el funcionamiento del algoritmo de cálculo de atributos durante el proceso de traducción de genotipo a fenotipo.

La gramática independiente del contexto que genera las cadenas del ejemplo que nos ocupa se define como $G = \{\Sigma_N, \Sigma_T, P, S\}$, donde el conjunto de símbolos no terminales Σ_N es:

$$\Sigma_N = \{cadena, tramas, trama, binario, numero, decimal\}$$

el conjunto Σ_T de terminales es,

$$\Sigma_T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

y el axioma,

$$S = cadena$$

Por último, el conjunto de reglas P contiene los siguientes elementos:

(0) <*cadena*> → <*numero*><*tramas*>

(0) <*tramas*> → <*trama*>
(1) | <*trama*><*tramas*>

(0) <*trama*> → <*binario*>
(1) | <*binario*><*trama*>

(0) <*binario*> → 0
(1) | 1

(0) <*numero*> → <*decimal*>
(1) | <*numero*><*decimal*>

(0)	<*decimal*>	→	0
(1)		|	1
(2)		|	2
(3)		|	3
(4)		|	4
(5)		|	5
(6)		|	6
(7)		|	7
(8)		|	8
(9)		|	9

La numeración independiente para cada no terminal tiene como objetivo simplificar la exposición del algoritmo de traducción de genotipo a fenotipo.

El sistema de atributos diseñado para comprobar las restricciones semánticas en el proceso de construcción del fenotipo se describe a continuación. En primer lugar se define el atributo *valor* asociado al símbolo no terminal <*numero*> para almacenar el valor numérico del no terminal que en la gramática se define como secuencia de dígitos decimales. Por otra parte, el símbolo <*decimal*> tiene un atributo también llamado *valor* que almacena el valor numérico del dígito. Ambos atributos son sintetizados por pertenecer al símbolo de la parte izquierda de la regla.

Las acciones semánticas para evaluar el atributo sintetizado *valor* del símbolo <*decimal*> son las siguientes:

<*decimal*>	→	0	{ <*decimal*> .*valor* = 0 }
	|	1	{ <*decimal*> .*valor* = 1 }
	|	2	{ <*decimal*> .*valor* = 2 }
	|	3	{ <*decimal*> .*valor* = 3 }
	|	4	{ <*decimal*> .*valor* = 4 }
	|	5	{ <*decimal*> .*valor* = 5 }
	|	6	{ <*decimal*> .*valor* = 6 }
	|	7	{ <*decimal*> .*valor* = 7 }
	|	8	{ <*decimal*> .*valor* = 8 }
	|	9	{ <*decimal*> .*valor* = 9 }

La evaluación del atributo *valor* del símbolo <*numero*> se realiza en las reglas correspondientes al no terminal de la siguiente manera:

$<numero> \rightarrow <decimal>$
{
$<numero>.valor = <decimal>.valor$
}
$<numero>_i \rightarrow <numero>_d <decimal>$
{
$<numero>_i.valor = <numero>_d.valor * 10 + <decimal>.valor$
}

En las acciones semánticas se evalúan los atributos de los símbolos y dichos valores se utilizan para realizar comprobaciones relativas a las restricciones semánticas del problema con el objetivo de rechazar a los individuos que no se ajusten a las condiciones semánticas establecidas. En particular, en el ejemplo que se está estudiando, el número natural que define la cantidad de tramas de la cadena debe ser mayor que 0. Se puede comprobar que se cumple esta restricción en la regla correspondiente al axioma consultando el atributo *valor* del no terminal $<numero>$ de la siguiente manera:

$<cadena> \rightarrow <numero><tramas>$
{
SI $(<numero>.valor == 0)$ **ERROR SEMÁNTICO**
}

En EGA, durante el proceso de construcción del fenotipo a partir del genotipo, se construye el árbol de derivación y se evalúan los atributos según el algoritmo propuesto de manera que, si se detecta un error semántico, se interrumpe el proceso y como consecuencia de ello se rechaza el individuo en construcción.

Una vez obtenido el valor que define el número correcto de tramas de la cadena, y comprobado que es estrictamente mayor que cero, se utiliza para controlar si efectivamente el número de tramas generadas coincide con dicho valor y si no es así rechazar el individuo en cuestión. Para ello se define el atributo *contador* asociado al símbolo $<tramas>$. Este atributo representa el número de tramas "pendientes" de generar en un instante dado del proceso de construcción del fenotipo. Se inicializa en la regla del axioma con el contenido del atributo *valor* del no terminal $<numero>$ de la siguiente manera:

$<cadena> \rightarrow <numero><tramas>$
{
    $<tramas>.contador = <numero>.valor$
}

El atributo *contador* entra en la categoría de atributos heredados entre hermanos, en concreto atributos heredados de hermanos situados a la izquierda que son los únicos que tienen sentido en el algoritmo propuesto. Para mantener actualizado dicho atributo, es decir, que en todo momento durante el proceso de generación de la cadena refleje el número de tramas "pendientes" de derivar, se debe decrementar cada vez que se derive una nueva trama. Para ello se incorpora la siguiente acción semántica:

$<tramas>_i \rightarrow <trama><tramas>_d$
{
    $<tramas>_d.contador = <tramas>_i.contador - 1$
}

Una vez diseñadas las acciones que permiten mantener actualizado el valor del atributo *contador* se diseñan las acciones que permiten detectar cadenas semánticamente incorrectas, que son aquellas en las que no coincide el número inicial de la cadena con la cantidad de tramas de la misma. La última trama de la cadena se genera con la primera producción del no terminal $<tramas>$, que es:

$$<tramas> \rightarrow <trama>$$

Dado el recorrido del árbol realizado por el algoritmo, para que la cadena en construcción sea semánticamente correcta, en la regla anterior, el valor del atributo *contador* del símbolo $<tramas>$ tiene que ser igual a 1, ya que si no es así, la cadena contiene menos tramas que las que debería tener. La acción semántica sería:

$<tramas> \rightarrow <trama>$
{
    SI $(<tramas>.contador \; ! = 1)$ **ERROR SEMÁNTICO**
}

Todas las tramas binarias excepto la última se derivan con la segunda producción del no terminal $<tramas>$, que es:

$$<tramas> \rightarrow <trama><tramas>$$

Por lo tanto para que se verifique la restricción semántica relativa a la igualad entre el número inicial de la cadena y la cantidad de tramas de la misma, en la producción anterior se debe cumplir que el atributo *contador* del símbolo de la parte izquierda valga como mínimo 2, es decir, el número de tramas "pendientes" de derivar es como mínimo 2, una se derivaría con el no terminal $<trama>$ y la(s) otra(s) con el no terminal $<tramas>$. La comprobación se puede hacer con la siguiente condición :

$<tramas>_i \quad \rightarrow \quad <trama><tramas>_d$
{
SI $(<tramas>_i .contador \ < 2)$ **ERROR SEMÁNTICO**
}

Por último, en cuanto a la componente "información global" de la gramática de atributos, en este ejemplo, no se necesita información global ya que las restricciones semánticas se pueden comprobar a partir de los atributos definidos para los símbolos.

A continuación se muestra la gramática de atributos completa del ejemplo.

Se define la gramática de atributos $G_A = \{\Sigma_T, \Sigma_N, S, P, K\}$, donde el conjunto de terminales Σ_T es:

$$\Sigma_T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Ninguno de los símbolos terminales tiene atributos definidos y por este motivo, para simplificar la notación, se omiten los paréntesis que acompañan a cada símbolo en la definición formal de una gramática de atributos.

El conjunto de los símbolos no terminales Σ_N es define como:

$$\begin{aligned} \Sigma_N = \{ \quad & cadena, \\ & tramas\ (\mathbb{N} \quad contador), \\ & trama, \\ & binario, \\ & numero\ (\mathbb{N} \quad valor), \\ & decimal\ (\mathbb{N} \quad valor) \quad \} \end{aligned}$$

De la misma manera que en el conjunto de los símbolos terminales, se han omitido los paréntesis que encierran la definición de los atributos para aquellos símbolos que no tienen atributos asociados.

El axioma de la gramática de atributos es:

$$S = cadena$$

El conjunto P de producciones está formado por las siguientes reglas con sus acciones semánticas:

(0) <*cadena*> → <*numero*><*tramas*>
{
SI (<*numero*> .*valor* == 0) **ERROR SEMÁNTICO**
<*tramas*> .*contador* =<*numero*> .*valor*
}

(0) <*tramas*> → <*trama*>
{
SI (<*tramas*> .*contador* ! = 1) **ERROR SEMÁNTICO**
}

(1) $<tramas>_i$ → <*trama*>$<tramas>_d$
{
SI ($<tramas>_i$.*contador* < 2) **ERROR SEMÁNTICO**
$<tramas>_d$.*contador* =$<tramas>_i$.*contador* − 1
}

(0) <*trama*> → <*binario*>

(1) <*trama*> → <*binario*><*trama*>

(0) <*binario*> → 0

(1) <*binario*> → 1

(0) <*numero*> → <*decimal*>
{
<*numero*> .*valor* =<*decimal*> .*valor*
}

(1) $<numero>_i$ → $<numero>_d$<*decimal*>
{
$<numero>_i$.*valor* =$<numero>_d$.*valor* ∗ 10+ <*decimal*> .*valor*
}

(0) <*decimal*> → 0 { <*decimal*> .*valor* = 0 }

(1) <*decimal*> → 1 { <*decimal*> .*valor* = 1 }

(2) <*decimal*> → 2 { <*decimal*> .*valor* = 2 }

(3) <*decimal*> → 3 { <*decimal*> .*valor* = 3 }

(4) <*decimal*> → 4 { <*decimal*> .*valor* = 4 }

(5) <*decimal*> → 5 { <*decimal*> .*valor* = 5 }

(6) <*decimal*> → 6 { <*decimal*> .*valor* = 6 }

(7) <*decimal*> → 7 { <*decimal*> .*valor* = 7 }

(8) <*decimal*> → 8 { <*decimal*> .*valor* = 8 }

(9) <*decimal*> → 9 { <*decimal*> .*valor* = 9 }

Por último, la gramática de atributos no tiene información global, es decir, $K = \Phi$.

En el ejemplo que nos ocupa, supongamos que el genotipo de partida para generar el fenotipo es el siguiente:

14	33	121	5	42	47	8	100	4	11	77

1	20	223	6	5	13	50	36	0	67	21

El genotipo está compuesto por 22 codones y, sólo por razones de espacio, se representa en dos tablas. La primera de ellas contiene los once primeros codones y la segunda los once últimos.

El algoritmo de transformación de genotipo a fenotipo de EGA comienza construyendo el nodo del árbol con el axioma de la gramática. El proceso continúa con la expansión del nodo situado más a la izquierda, que en este paso del algoritmo es precisamente la raíz, y como el axioma de la gramática sólo tiene una producción, no se consume ningún codón del genotipo. Aplicando la regla 1a) del algoritmo, se crean los nodos de los símbolos de la parte derecha de la regla y se asignan como hijos al nodo expandido manteniendo su orden. En la figura 5.1 se muestra el estado actual del árbol de derivación. Se puede observar que cada nodo del árbol se representa con un círculo que encierra el símbolo correspondiente y un rectángulo para el(los) atributos(s) definidos para el símbolo en la gramática de atributos. La producción del axioma no contiene en su acción semántica ningún cálculo relativo a atributos heredados, que son precisamente los que se evalúan en el proceso de expansión de un nodo, y por lo tanto no procede la aplicación de la regla 1b). Por este motivo las cajas dibujadas en los nodos hijos del nodo raíz aparecen vacías ya que carecen de valor en este paso del algoritmo. Además, cada nodo contiene una elipse interna con un número que corresponde al codón utilizado para expandir dicho nodo. En el caso de que no se consuma ningún codón en la expansión del nodo, la elipse aparecerá de color negro. Por último, en cada árbol de derivación, se marcará el último nodo expandido con una estrella en su interior.

El algoritmo continúa con el nodo pendiente de expandir, más profundo y situado más a la izquierda, que es el correspondiente al símbolo <*numero*>, y el codón que va a determinar la regla para expandir vale 14. Como $14 \; mod \; 2 = 0$ se utiliza la producción (0) del no terminal <*numero*> para realizar la expansión aplicando la regla 1a) del algoritmo y obtener el árbol

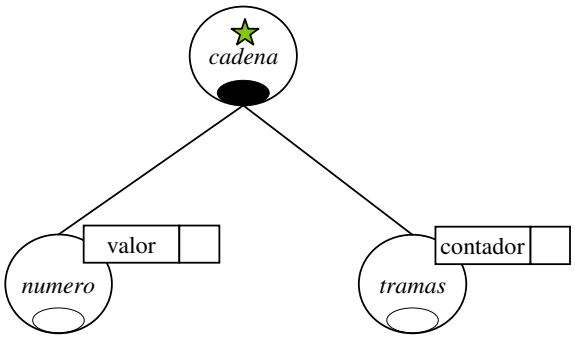

Figura 5.1: Árbol de derivación de EGA después de la expansión del nodo raíz.

de derivación que se muestra en la figura 5.2. En la regla que se ha utilizado para expandir el nodo no hay cálculo de atributos heredados y por lo tanto en la expansión del nodo únicamente se crean los nodos hijos.

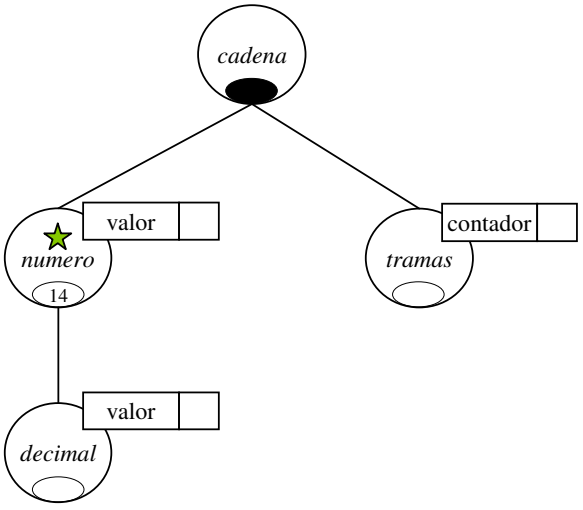

Figura 5.2: Árbol de derivación de EGA después de la expansión del nodo del símbolo <*numero*> .

El siguiente nodo a expandir es el correspondiente al símbolo no terminal <*decimal*> y el codón actual es 33. Dicho no terminal tiene 10 reglas, lo que hace que la regla que se utilice para la expansión sea la regla (3) ya que $33 \; mod \; 10 = 3$. Aplicando la regla 1a) del algoritmo se expande el nodo dando lugar al árbol que se muestra en la figura 5.3. La producción utilizada para la

expansión no incluye en su acción semántica el cálculo de atributos heredados y por lo tanto el proceso de expansión termina con la creación del nodo hijo. En la figura 5.3 se puede observar que el nodo correspondiente al terminal 3 aparece sombreado, el objetivo es distinguir los nodos hoja correspondientes a símbolos terminales de los nodos intermedios asociados a los no terminales de la gramática. La regla 2 del algoritmo establece que cuando se crea un nodo hoja se termina su procesamiento, y por lo tanto después de la creación del nodo del terminal 3 se desencadenan las tareas asociadas a la terminación del nodo especificadas en las reglas 2a) y 2b). Por una parte, no procede aplicar la regla 2a) porque es aplicable en la terminación de un nodo con hermanos y el nodo del símbolo 3 no tiene hermanos. Por el contrario si es aplicable la regla 2b) que define que en el momento en el que termina el procesamiento de todos los nodos hijos de un nodo padre se evalúan los atributos sintetizados si los hubiera. En concreto, el símbolo <*decimal*> tiene un atributo sintetizado, el atributo *valor* y se evalúa en este paso del algoritmo según se especifica en la acción semántica de la producción (3) de dicho símbolo:

$$(3) \quad <decimal> \rightarrow 3 \qquad \{ <decimal>.valor = 3 \}$$

En la figura 5.3 queda reflejado el valor del atributo *valor* del símbolo <*decimal*>.

La terminación del nodo del símbolo <*decimal*> implica la terminación de su nodo padre, el correspondiente al no terminal <*numero*> ya que es el único hijo. Se aplica la regla 2b) concerniente a la evaluación de atributos sintetizados para evaluar el atributo sintetizado *valor* del símbolo <*numero*> aplicando la acción semántica siguiente:

$$(0) \quad <numero> \rightarrow <decimal> \qquad \{ <numero>.valor = <decimal>.valor \}$$

En la figura 5.3 queda reflejado el resultado de la evaluación del atributo *valor* del símbolo <*numero*>.

El nodo del símbolo <*numero*> es el primer hijo de la raíz del árbol, y su terminación desencadena la aplicación de la regla 2a) relativa a la evaluación de atributos heredados entre hermanos. En concreto, el segundo hijo del nodo raíz que corresponde al no terminal $< tramas >$ tiene el atributo *contador* heredado de su hermano izquierdo que se evalúa según se describe en la acción semántica del axioma:

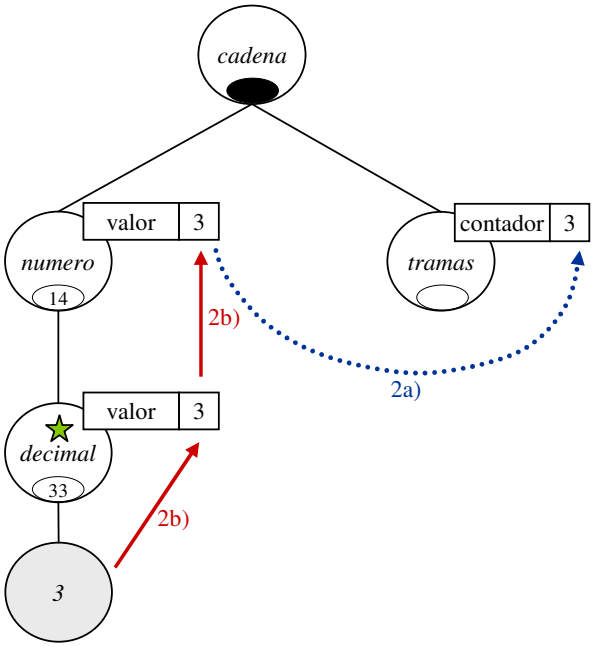

Figura 5.3: Árbol de derivación de EGA después de la expansión del nodo del símbolo <*decimal*> .

(0) <*cadena*> → <*numero*><*tramas*>
{
SI (<*numero*> .*valor* == 0) **ERROR SEMÁNTICO**
<*tramas*> .*contador* =<*numero*> .*valor*
}

Este es el primer instante en el que la gramática de atributos diseñada para el problema incorpora la detección de un error semántico. Según la regla 5) del algoritmo, la detección de un error semántico interrumpe el proceso de traducción y rechaza al individuo cuyo fenotipo se está construyendo. En concreto, como se puede ver en la acción semántica del axioma, si el atributo *valor* del símbolo <*numero*> valiera 0 se produciría un error semántico ya que una cadena con cero tramas binarias es incorrecta semánticamente. En el ejemplo que nos ocupa esta condición no es cierta ya que el atributo *valor* del símbolo <*numero*> vale 3, y por lo tanto continúa el proceso de generación del fenotipo.

El siguiente nodo que el algoritmo selecciona para expandir es el correspondiente al no terminal <*tramas*> y el codón actual vale 121. Dado

que el símbolo <*tramas*> tiene dos producciones, se utilizará la producción (1) ya que 121 *mod* 2 = 1. Aplicando la regla 1a) del algoritmo se añaden al árbol los nodos hijos. También se aplica la regla 1b) dado que existe herencia de atributos, en concreto, el segundo hijo, que corresponde al no terminal <*tramas*>, hereda de su padre el atributo *contador* decrementado en uno. En la figura 5.4 se muestra el estado del árbol de derivación después de la última operación de expansión. En la gramática de atributos se puede observar que en la acción semántica de la producción (1) del no terminal $< tramas >$, antes del cálculo de los atributos, aparece una sentencia de control semántico que descarta el fenotipo en construcción si el valor del atributo *contador* del símbolo de la izquierda de la producción es menor que 2:

(1) $<tramas>_i \rightarrow <trama><tramas>_d$
{
SI ($<tramas>_i$.*contador* < 2) **ERROR SEMÁNTICO**
$<tramas>_d$.*contador* =$<tramas>_i$.*contador* $- 1$
}

Efectivamente, si ese valor es menor que 2 se estaría generando un fenotipo con más tramas binarias que las que indica el primer número de la cadena, y por lo tanto sería un fenotipo semánticamente incorrecto.

El siguiente nodo por expandir, el situado más a la izquierda de los más profundos, es el correspondiente al no terminal <*trama*> y el valor del codón actual es 5. Por lo tanto, la producción para la expansión del nodo es la (1) ya que el símbolo no terminal <*trama*> tiene dos partes derechas y 5 *mod* 2 = 1. Se aplica la regla 1a) del algoritmo referente a la incorporación al árbol de los nodos hijos del nodo en expansión y se obtiene el árbol que se muestra en la figura 5.5. La regla otra regla relacionada con el proceso de expansión de los nodos, la 1b), no se aplica porque no existen atributos heredados.

El algoritmo continúa con la expansión del nodo correspondiente al símbolo <*binario*> y el codón 42. La aplicación de la regla 1a) con la producción (0) del no terminal <*binario*> da lugar al árbol de derivación que se muestra en la figura 5.6. En cuanto a la aplicación de la regla 1b), en este paso del algoritmo no procede ya que no hay atributos heredados por los hijos, en este caso el único hijo, del nodo en expansión. La creación del nodo del terminal 0 según la regla 2) implica su terminación por ser un nodo hoja y, además, al ser el único hijo del nodo correspondiente al no terminal <*binario*> implica también la terminación de éste. Si el símbolo <*binario*> tuviera algún atributo sintetizado se calcularía en este momento, pero la gramática de atributos para este ejemplo no define atributos sintetizados para este símbolo. La ter-

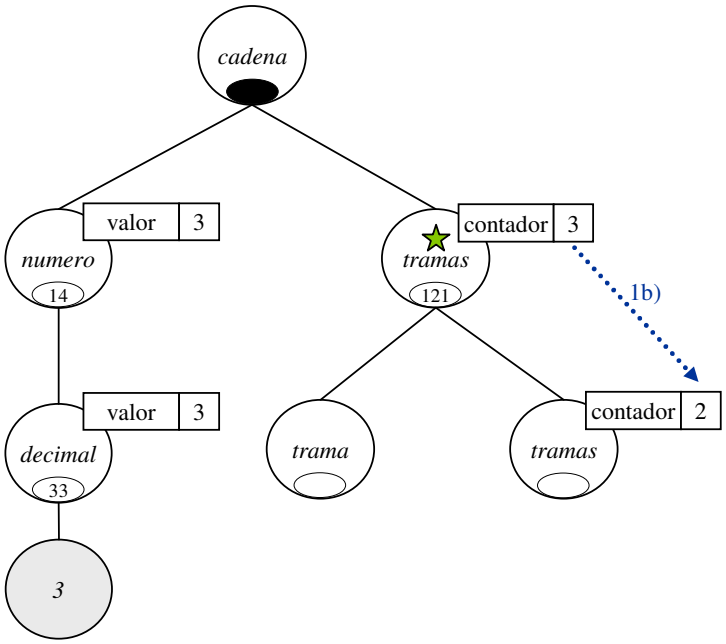

Figura 5.4: Árbol de derivación de EGA después de la expansión del nodo del símbolo <*tramas*> en el primer nivel del árbol.

minación del nodo del símbolo <*binario*>, al ser el primer hijo de su nodo padre, podría desencadenar el cálculo de atributos heredados entre hermanos si estuvieran definidos y, en concreto, en este ejemplo, no lo están. De esta manera el algoritmo continúa con la expansión del nodo correspondiente al no terminal <*trama*> que aún no se ha expandido.

En la gramática de atributos se puede observar que las producciones relacionadas con la generación de cada trama binaria carecen de acciones semánticas para la evaluación de atributos y control semántico. Por ello, se omitirá la descripción paso a paso del algoritmo en las fases de generación de las tramas binarias. En la figura 5.7 se muestra el estado del árbol de derivación con el subárbol completo de la primera trama binaria habiéndose consumido para su generación los codones 47, 8, 100 y 4. En las figuras posteriores a la figura 5.7, los subárboles correspondientes a las tramas binarias aparecerán representados simbólicamente por un triángulo con las hojas del subárbol en la base en lugar de la composición completa de todos sus nodos.

Cuando termina la generación de la primera trama binaria, el algoritmo continúa con la expansión del nodo correspondiente a la segunda aparición del no terminal $<tramas>$ y el codón 11. Como el símbolo $<tramas>$ tiene dos producciones, se utiliza para la expansión la producción (1) ya que

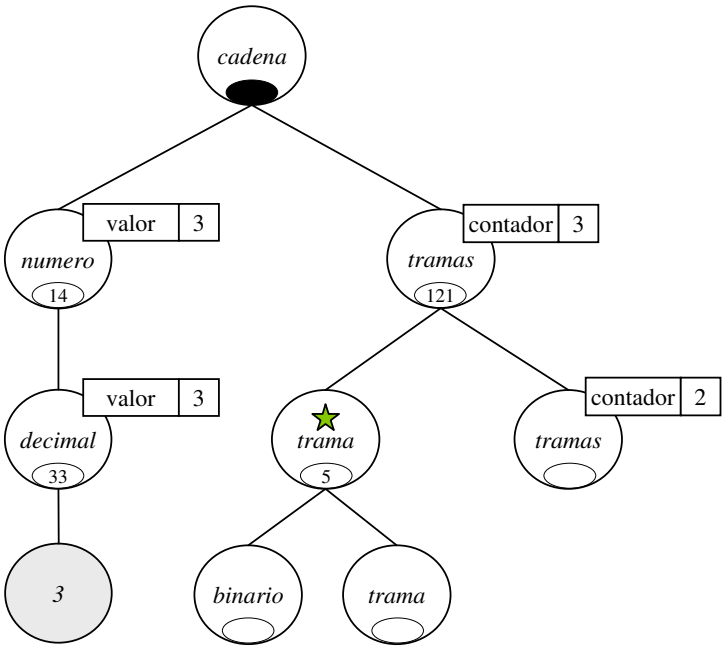

Figura 5.5: Árbol de derivación de EGA después de la expansión del nodo del símbolo <*trama*>

11 *mod* 2 = 1. Como resultado de la aplicación de la regla 1a) del algoritmo se añaden al árbol de derivación los dos nodos hijos del nodo en expansión como se observa en la figura 5.8.

La producción usada en la expansión del último nodo tiene asociada la siguiente acción semántica:

(1) $<tramas>_i \rightarrow <trama><tramas>_d$
{
SI ($<tramas>_i$.*contador* < 2) **ERROR SEMÁNTICO**
$<tramas>_d$.*contador* =$<tramas>_i$.*contador* $- 1$
}

Como puede observarse, la acción semántica incluye tareas de control semántico y tareas de cálculo de atributos heredados de padre a hijo. El control semántico, como ya se explicó anteriormente, compara con 2 el contenido del atributo *contador* del símbolo de la parte izquierda de la producción ya que, si ese valor es menor que 2, se estaría generando un fenotipo con más tramas binarias que las que indica el primer número de la cadena, y por lo tanto sería un fenotipo semánticamente incorrecto. En cuanto al cálculo de

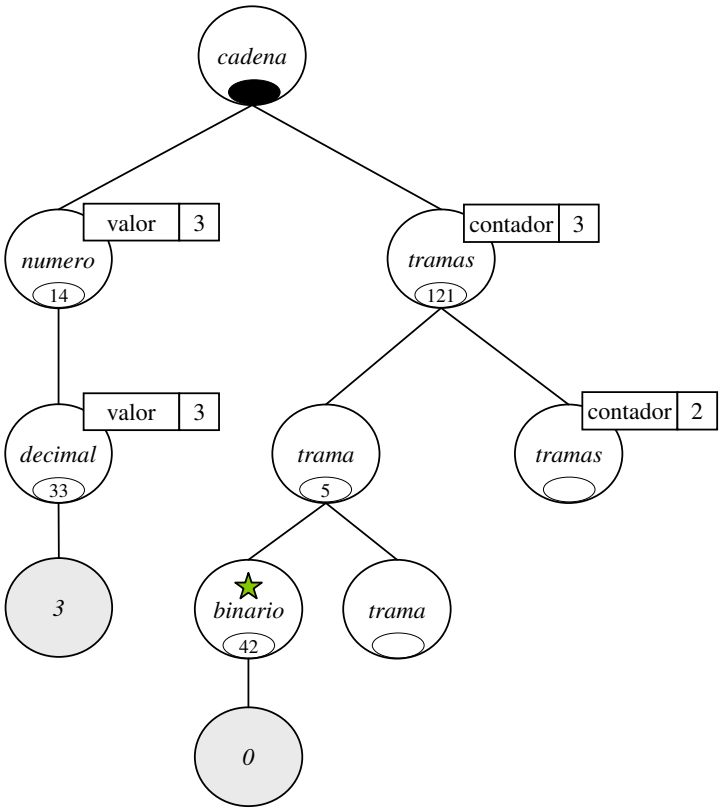

Figura 5.6: Árbol de derivación de EGA después de la expansión del nodo del símbolo <*binario*>

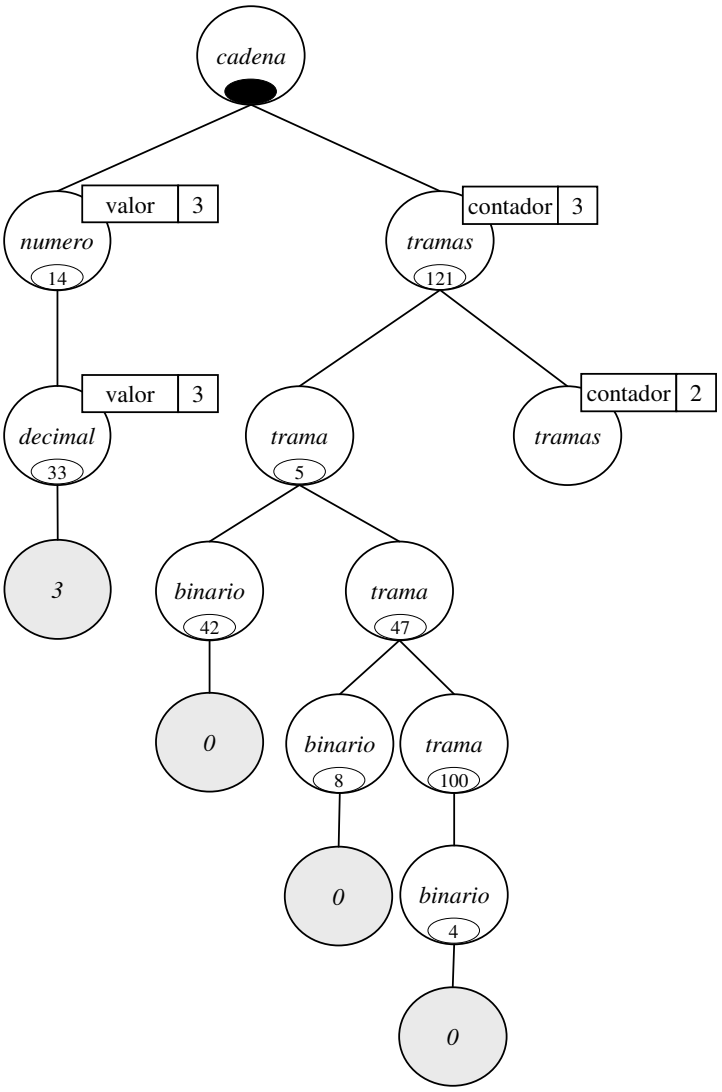

Figura 5.7: Árbol de derivación de EGA después de la expansión de la primera trama binaria

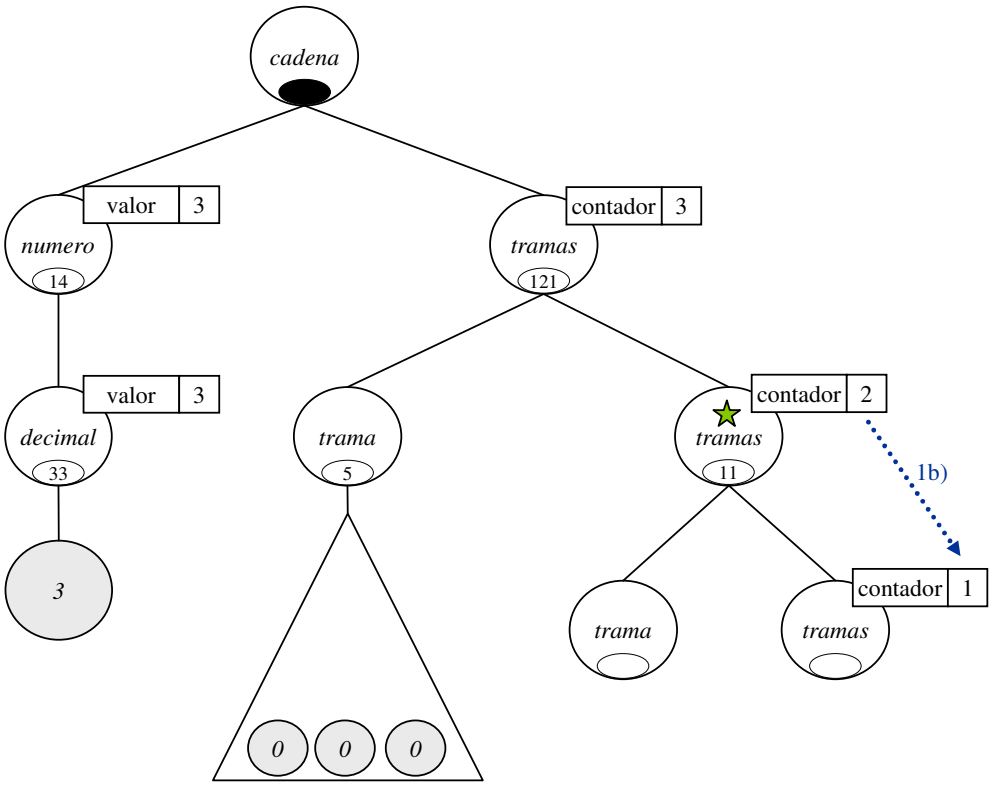

Figura 5.8: Árbol de derivación de EGA después de la expansión de la segunda aparición del símbolo <*tramas*>.

atributos, el segundo hijo que corresponde al no terminal <*tramas*> hereda de su padre el atributo *contador* decrementado en uno como puede observarse en la figura 5.8.

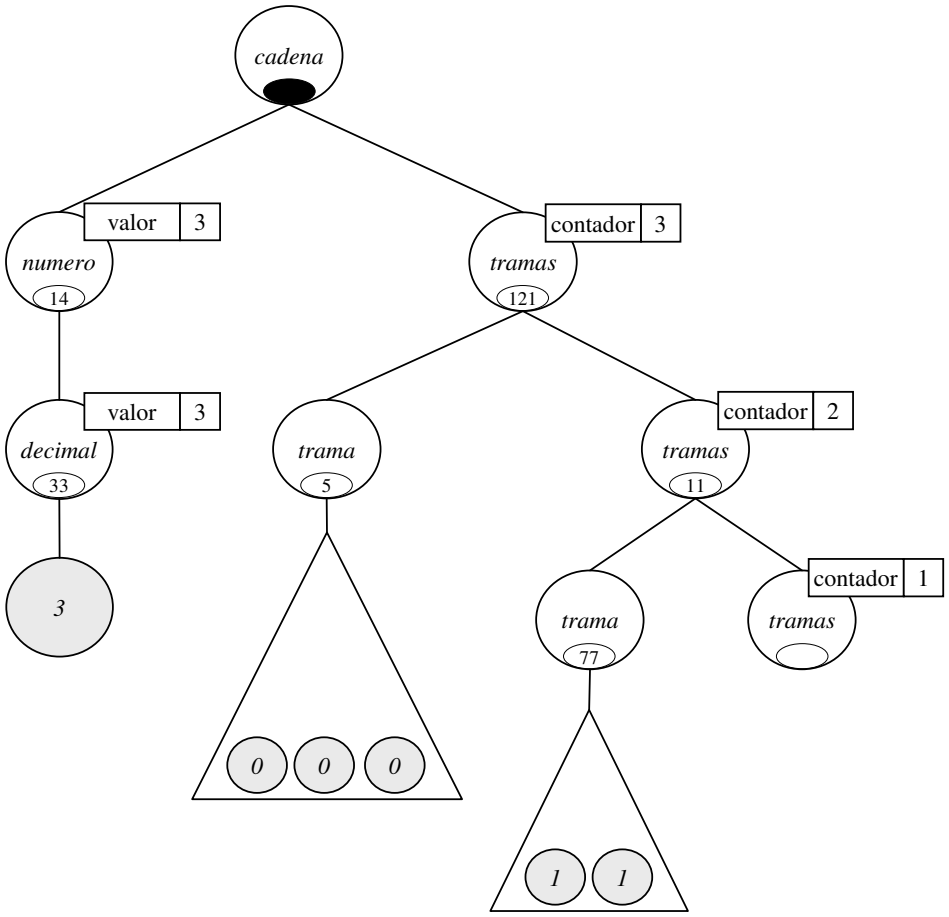

Figura 5.9: Árbol de derivación de EGA después de la expansión de la segunda trama binaria.

El algoritmo continúa con la segunda aparición del nodo del no terminal <*trama*>. Se omite la explicación de la evolución del algoritmo durante la generación de la trama binaria porque no se ejecuta ninguna acción semántica, y el algoritmo se reduce a la expansión de los nodos del subárbol. En resumen, tras la aplicación de las producciones determinadas por los cuatro codones siguientes, 77, 1, 20, y 223 se genera la trama binaria "1 1"y el árbol de derivación queda como se muestra esquemáticamente en la figura 5.9.

El proceso continúa con la expansión del siguiente nodo, que es el correspondiente a la tercera aparición del símbolo $< tramas >$. El codón

actual vale 6 y determina la aplicación de la regla (0) del no terminal $<tramas>$ ya que $6 \; mod \; 2 = 0$. Dicha regla tiene el siguiente aspecto:

(0) $<tramas> \rightarrow <trama>$
{
SI ($<tramas>$ *.contador* $! = 1$) **ERROR SEMÁNTICO**
}

Como ya se explicó en el diseño de la gramática de atributos, dado que la producción (0) del símbolo $< tramas >$ corresponde a la última trama de la cadena, es imprescindible para garantizar la corrección semántica, que el atributo *contador* del símbolo de la parte izquierda de dicha producción valga 1. La regla 5) del algoritmo establece la interrupción del proceso de construcción del fenotipo en el caso en el que se viole alguna de las condiciones establecidas sobre los atributos. En este ejemplo, se cumple que el atributo *contador* del símbolo $<tramas>$ correspondiente a la última trama vale 1, y por lo tanto, no es aplicable la regla 5) del algoritmo. De esta manera, el árbol de derivación quedaría como se muestra en la figura 5.10.

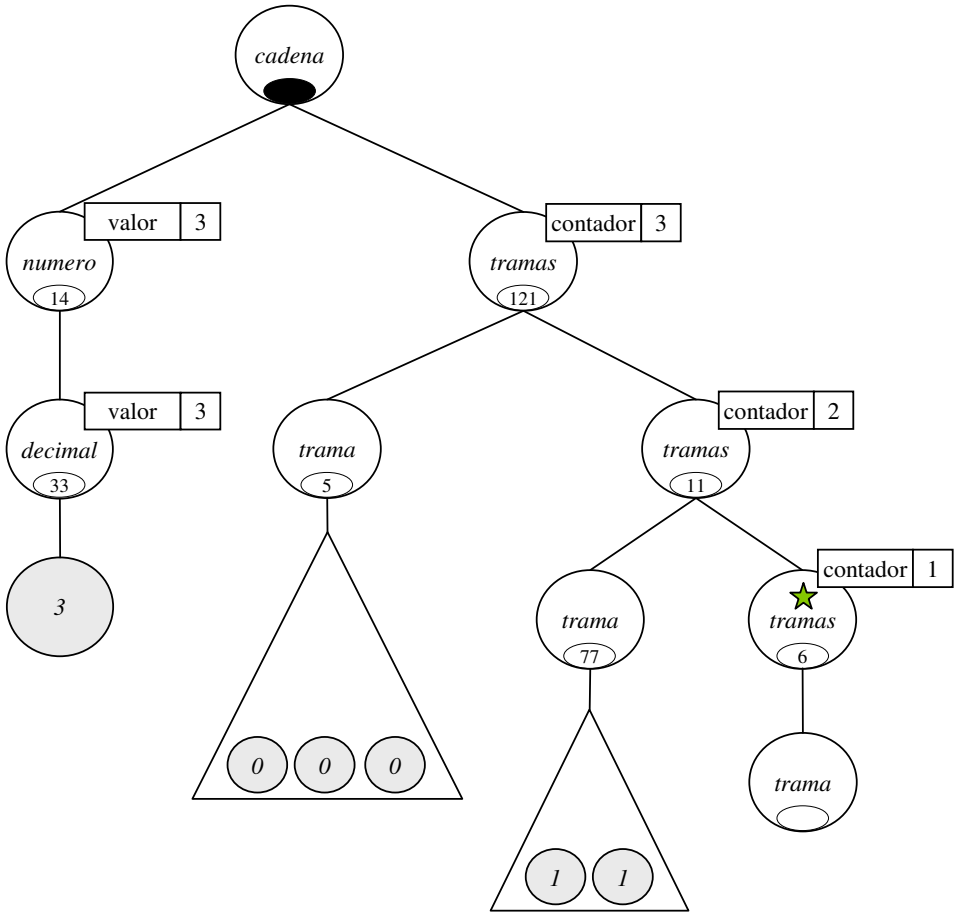

Figura 5.10: Árbol de derivación de EGA después de la expansión del último nodo del símbolo <*tramas*>.

El siguiente nodo a expandir corresponde a la tercera aparición del símbolo <*trama*>. El subárbol generado a partir de dicho símbolo con los siguientes cuatro codones, 5,13,50 y 36 da lugar a la última trama binaria "1 0". El la figura 5.11 se muestra el estado en el que queda el árbol de derivación después de la generación de la última trama binaria.

La terminación del nodo correspondiente a la tercera y última aparición del símbolo <*trama*> implica la terminación de su nodo padre, asociado a la tercera aparición del no terminal <*tramas*> y cuyo atributo *contador* vale 1. A su vez, la terminación de este último nodo hace que también termine el nodo correspondiente a la segunda aparición del símbolo <*tramas*>, la que tiene asignado un 2 en su atributo *contador*. Se repite el encadenamiento en la finalización de los nodos, de manera que al terminar el nodo de la segunda

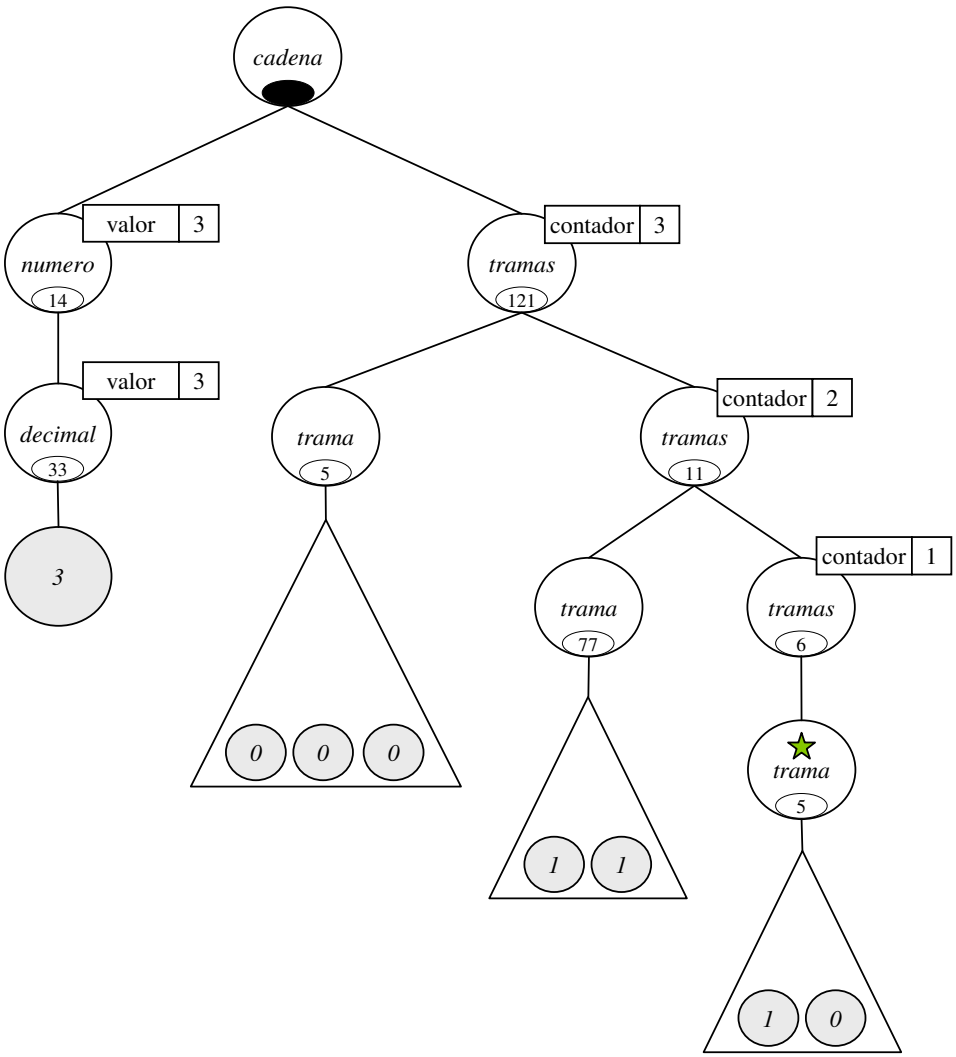

Figura 5.11: Árbol de derivación de EGA después de la expansión de la última trama binaria.

aparición de <*tramas*> se produce también la terminación de la primera aparición del mismo símbolo, aquella cuyo atributo *contador* vale 3. Y finalmente, como han terminado todos los hijos del nodo raíz, también termina este nodo. En este momento, según la regla 3) del algoritmo, se considera terminada la construcción del árbol de derivación con la correspondiente evaluación de los atributos de todos sus nodos y por lo tanto se puede considerar que el fenotipo generado es correcto sintáctica y semánticamente. El genotipo no ha sido consumido completamente en la generación del fenotipo, quedando los últimos 3 codones sin utilizar.

En el ejemplo que se ha utilizado como soporte para la explicación del algoritmo de generación del genotipo a partir del fenotipo en EGA, han aparecido los diferentes casos de evaluación de atributos que contempla el algoritmo, es decir, evaluación de atributos heredados por la izquierda (de padre o hermano) y atributos sintetizados. Además el ejemplo fue diseñado para que el fenotipo obtenido fuera semánticamente correcto. A continuación se presentan tres casos en los que los fenotipos que se construyen son semánticamente incorrectos. Se han diseñado tres casos porque la gramática de atributos del ejemplo contempla tres situaciones distintas en las que se produce un error semántico.

El primer caso muestra la construcción de un fenotipo que es incorrecto porque el número natural situado al principio de la cadena es cero. Supóngase un genotipo cuyos dos primeros codones son 144 y 10. La ejecución del algoritmo utilizando éstos codones se puede resumir en los siguientes pasos:

1. Construcción de la raíz del árbol con el símbolo <*cadena*> que corresponde al axioma de la gramática.

2. Expansión del nodo raíz sin consumo de codón que añade al árbol los nodos de los símbolos <*numero*> y <*tramas*>.

3. Expansión del nodo del no terminal <*numero*> con el codón 144 que añade al árbol el nodo del símbolo <*decimal*>.

4. Expansión del nodo del no terminal <*decimal*> con el codón 10 que añade al árbol del nodo del símbolo 0.

5. Terminación del nodo del terminal 0 y evaluación del atributo sintetizado *valor* del símbolo <*decimal*>. El valor asignado al atributo es 0.

6. Terminación del nodo del no terminal <*decimal*> y evaluación del atributo sintetizado *valor* del símbolo <*numero*>. El valor asignado al atributo es 0.

7. Terminación del nodo del no terminal <*numero*>.

8. Control semántico del atributo *valor* del no terminal <*numero*>. En este ejemplo se detecta un error semántico ya que el valor del número natural situado al principio de la cadena debe ser mayor que 0. El algoritmo finaliza con error semántico y el consecuente rechazo del individuo.

El árbol del derivación se muestra en la figura 5.12.

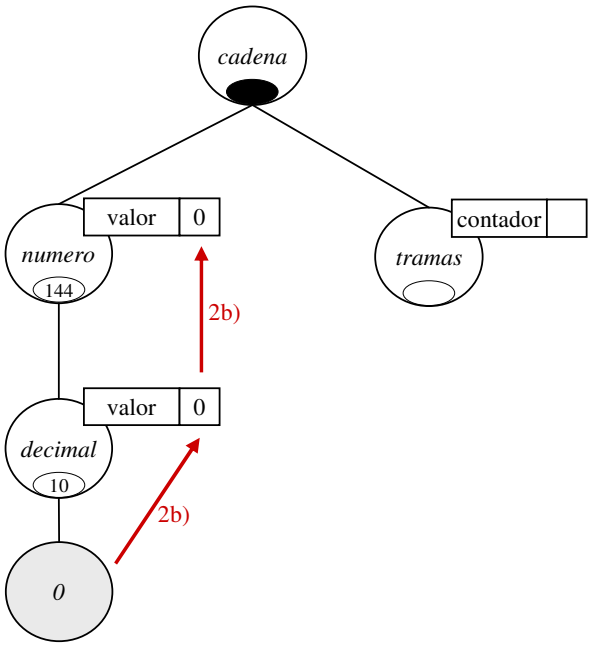

Figura 5.12: Árbol de derivación de EGA correspondiente a un fenotipo incorrecto.

El segundo caso de fenotipo incorrecto es el generado a partir del genotipo cuyos primeros codones son 2, 45 y 10. El fenotipo es incorrecto porque el número de tramas binarias de la cadena es menor que el valor del número natural situado al principio. En la figura 5.13 se muestra el árbol de derivación construido en la ejecución del algoritmo a partir de los tres codones mencionados.

Los pasos del algoritmo hasta su interrupción como consecuencia de un error semántico son los siguientes:

1. Construcción de la raíz del árbol con el símbolo <*cadena*> que corresponde al axioma de la gramática.

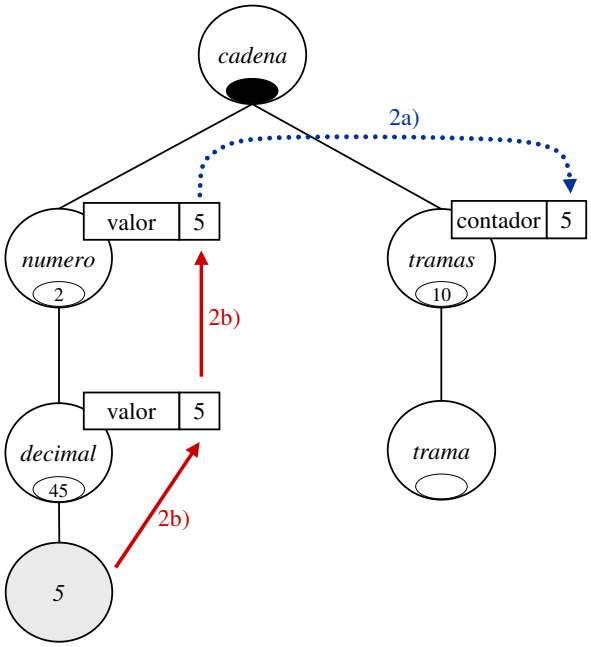

Figura 5.13: Árbol de derivación de EGA correspondiente a un fenotipo incorrecto.

2. Expansión del nodo del no terminal <*cadena*> sin consumo de codón que añade al árbol los nodos de los símbolos <*numero*> y <*tramas*>.

3. Expansión del nodo del no terminal <*numero*> con el codón 2 que añade al árbol el nodo del símbolo <*decimal*>.

4. Expansión del nodo del no terminal <*decimal*> con el codón 45 que añade al árbol del nodo del símbolo 5.

5. Terminación del nodo del terminal 5 y evaluación del atributo sintetizado *valor* del símbolo <*decimal*>. El valor asignado al atributo es 5.

6. Terminación del nodo del no terminal <*decimal*> y evaluación del atributo sintetizado *valor* del símbolo <*numero*>. El valor asignado al atributo es 5.

7. Terminación del nodo del no terminal <*numero*>.

8. Control semántico que termina con éxito del atributo *valor* del no terminal <*numero*>.

9. Evaluación del atributo *contador* del símbolo <*tramas*> a partir del atributo *valor* de su nodo hermano correspondiente al no terminal <*numero*>. El valor asignado al atributo es 5.

10. Expansión del nodo del no terminal <*tramas*> con el codón 10 que añade al árbol de derivación el nodo del símbolo <*trama*>. En este punto se realiza control semántico sobre el atributo *contador* del símbolo <*tramas*>, y termina con error ya que el valor del atributo debe ser 1 y en este caso vale 5. El algoritmo finaliza con error semántico y el consecuente rechazo del individuo.

El tercer caso tiene como objetivo mostrar el error semántico que ocurre cuando el número de tramas es mayor que el valor del número natural situado al principio de la cadena. El genotipo en este caso comienza por los codones 12, 11 y 25, y los correspondientes pasos del algoritmo son:

1. Construcción de la raíz del árbol con el símbolo <*cadena*> que corresponde al axioma de la gramática.

2. Expansión del nodo del no terminal <*cadena*> sin consumo de codón que añade al árbol los nodos de los símbolos <*numero*> y <*tramas*>.

3. Expansión del nodo del no terminal <*numero*> con el codón 12 que añade al árbol el nodo del símbolo <*decimal*>.

4. Expansión del nodo del no terminal <*decimal*> con el codón 11 que añade al árbol del nodo del símbolo 1.

5. Terminación del nodo del terminal 1 y evaluación del atributo sintetizado *valor* del símbolo <*decimal*>. El valor asignado al atributo es 1.

6. Terminación del nodo del no terminal <*decimal*> y evaluación del atributo sintetizado *valor* del símbolo <*numero*>. El valor asignado al atributo es 1.

7. Terminación del nodo del no terminal <*numero*>.

8. Control semántico que termina con éxito del atributo *valor* del no terminal <*numero*>.

9. Evaluación del atributo *contador* del símbolo <*tramas*> a partir del atributo *valor* de su nodo hermano correspondiente al no terminal <*numero*>. El valor asignado al atributo es 1.

10. Expansión del nodo del no terminal <*tramas*> con el codón 25 que añade al árbol de derivación los nodos de los símbolos <*trama*> y <*trama*>. En este punto se realiza control semántico sobre el atributo *contador* del símbolo <*tramas*> situado a la izquierda de la producción, y termina con error ya que el valor del atributo debe ser mayor o igual que dos y en este caso vale 1. El algoritmo finaliza con error semántico y el consecuente rechazo del individuo.

En la figura 5.14 se muestra el árbol de derivación correspondiente al último caso.

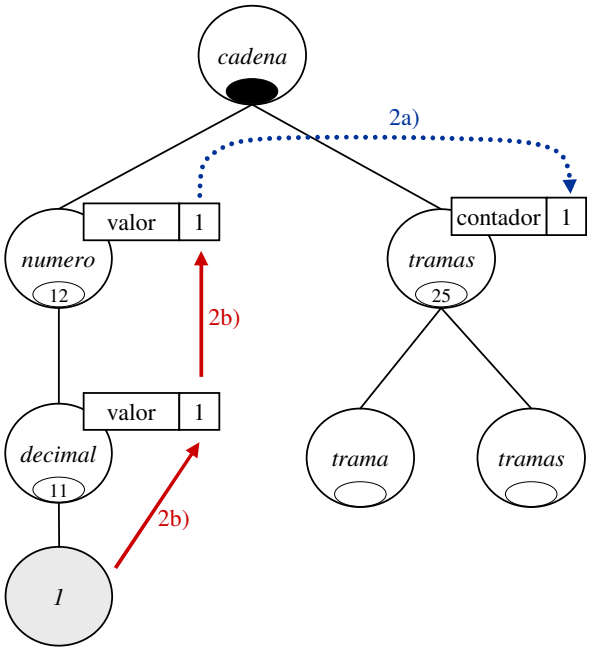

Figura 5.14: Árbol de derivación de EGA correspondiente a un fenotipo incorrecto.

Obviamente, el diseño de la gramática de atributos es un aspecto fundamental en la resolución de problemas con EGA. Por una parte, el objetivo primordial de la gramática de atributos es captar o representar todas las restricciones semánticas del problema concreto, y por otra, es deseable que el diseño de la misma permita detectar **lo más pronto posible** fenotipos semánticamente incorrectos. Este segundo aspecto está relacionado con el rendimiento de la técnica de EGA. Cuanto antes se detecte que un fenotipo es incorrecto, menos tiempo se invierte en su construcción, y en definitiva, si un fenotipo es incorrecto, va a ser rechazado y por lo tanto todo el tiempo invertido en él, es tiempo "perdido".

Por ejemplo, en el problema que se ha presentado previamente, en el que los fenotipos están compuestos de un número natural seguido de una serie de tramas binarias, la gramática de atributos propuesta permite capturar las restricciones semánticas del problema, y además, también garantiza la detección de fenotipos incorrectos lo más pronto posible. Veamos otra gramática de atributos que cumple su función principal, es decir, capturar las restricciones semánticas del problema, pero que no es óptima en términos de rendimiento ya que no permite detectar lo antes posible fenotipos semánticamente incorrectos.

En esta nueva gramática de atributos, que posteriormente se definirá de manera formal, la idea principal es contar las tramas que se generan durante la generación del fenotipo, y una vez terminado éste, comparar el número de tramas generadas con el valor representado por el número natural situado al principio de la cadena. Para ello, se asigna al axioma un atributo sintetizado que contenga el número de tramas que debe contener la cadena y por otra parte, en la definición recursiva de las tramas, se utiliza otro atributo sintetizado que se inicializa a 1 en la regla correspondiente al caso terminal de la recursividad, y se incrementa en 1 en la regla recursiva. De esta manera, durante la generación de las tramas se puede contar cuántas se han generado, y cuando se sintetice el axioma de la gramática, es decir, cuando termine de procesarse del nodo raíz del árbol de derivación, es posible comparar el número de tramas generadas con el número correcto de tramas que está almacenado en un atributo del axioma. Es evidente que con esta nueva gramática no es posible decidir si un fenotipo es o no semánticamente correcto hasta que no se termina su construcción, y por lo tanto, es menos eficiente que la gramática anterior que no exigía la terminación de la construcción de los fenotipos para decidir su validez semántica.

En definitiva, la nueva y poco eficiente gramática de atributos se define como $G_B = \{\Sigma_T, \Sigma_N, S, P, K\}$, donde el conjunto de terminales Σ_T es:

$$\Sigma_T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Como puede observarse, ninguno de los símbolos terminales tiene atributos definidos.

El conjunto de los símbolos no terminales Σ_N se define como:

$$\begin{aligned}\Sigma_N = \{ \quad & cadena\ (\mathbb{N} \quad total),\\ & tramas\ (\mathbb{N} \quad contador),\\ & trama,\\ & binario,\\ & numero\ (\mathbb{N} \quad valor),\\ & decimal\ (\mathbb{N} \quad valor) \quad \}\end{aligned}$$

De la misma manera que en el conjunto de los símbolos terminales, se han omitido los paréntesis que encierran la definición de los atributos para aquellos símbolos que no tienen atributos asociados.

El axioma de la gramática de atributos es:

$$S = cadena$$

El conjunto P de producciones está formado por las siguientes reglas con sus acciones semánticas:

(0) <*cadena*> → <*numero*><*tramas*>
{
SI (<*numero*> .*valor* == 0) **ERROR SEMÁNTICO**
<*cadena*> .*total* =<*numero*> .*valor*
SI (<*cadena*> .*total* ! = <*tramas*> .*contador*) **ERROR SEMÁNTICO**
}

(0) <*tramas*> → <*trama*>
{
<*tramas*> .*contador* = 1
}

(1) <*tramas*>$_i$ → <*trama*><*tramas*>$_d$
{
<*tramas*>$_i$.*contador* =<*tramas*>$_d$.*contador* + 1
}

(0) <*trama*> → <*binario*>

(1) <*trama*> → <*binario*><*trama*>

(0) <*binario*> → 0

(1) <*binario*> → 1

(0) <*numero*> → <*decimal*>
{
<*numero*> .*valor* =<*decimal*> .*valor*
}

(1) <*numero*>$_i$ → <*numero*>$_d$<*decimal*>
{
<*numero*>$_i$.*valor* =<*numero*>$_d$.*valor* ∗ 10+ <*decimal*> .*valor*
}

(0) <*decimal*> → 0 { <*decimal*> .*valor* = 0 }
(1) <*decimal*> → 1 { <*decimal*> .*valor* = 1 }
(2) <*decimal*> → 2 { <*decimal*> .*valor* = 2 }
(3) <*decimal*> → 3 { <*decimal*> .*valor* = 3 }
(4) <*decimal*> → 4 { <*decimal*> .*valor* = 4 }
(5) <*decimal*> → 5 { <*decimal*> .*valor* = 5 }
(6) <*decimal*> → 6 { <*decimal*> .*valor* = 6 }
(7) <*decimal*> → 7 { <*decimal*> .*valor* = 7 }
(8) <*decimal*> → 8 { <*decimal*> .*valor* = 8 }
(9) <*decimal*> → 9 { <*decimal*> .*valor* = 9 }

Por último, no se define información global de la gramática de atributos, es decir, $K = \Phi$.

5.3. Primer ejemplo: regresión simbólica

El primer ejemplo que se plantea para aplicar el método de evolución gramatical con semántica se selecciona porque aparece en [O'Neill y Ryan (2003)] y además está descrito con más detalle en [Koza (1992)]. Se trata del problema de la regresión simbólica. En los próximos apartados se describe en qué consiste el problema, se plantea su resolución haciendo uso de EG y por último se proponen dos soluciones distintas con EGA, utilizando diferentes restricciones semánticas en cada una de las propuestas.

5.3.1. Planteamiento del problema

Este primer ejemplo es un problema clásico, ya resuelto por Koza [Koza (1992)] mediante algoritmos de programación genética, y posteriormente utilizando EG en [O'Neill y Ryan (2003)]. El problema consiste en encontrar la expresión matemática de una función que se ajuste a un conjunto predeterminado de puntos. En concreto, la función que se busca es:

$$f(x) = x^4 + x^3 + x^2 + x$$

y el conjunto de puntos de ajuste pertenece al intervalo [-1,+1].

5.3.2. Solución del problema con EG

En primer lugar se resuelve el problema utilizando la versión ajustada de EG descrita en el capitulo 4 de este trabajo. Precisamente, el intento de reproducir este ejemplo de regresión simbólica, entendiendo que la reproducción

tenía éxito si se obtenían rendimientos similares a los descritos en [O'Neill y Ryan (2003)], permitió aclarar la confusión que en un principio suscitaron las gráficas de rendimiento que aparecen en [O'Neill y Ryan (2003)]. En el capítulo 4 se describe en profundidad este aspecto.

El primer paso consiste en definir los dos módulos del sistema. Por una parte la gramática independiente del contexto, y por otra los valores concretos de los elementos configurables del algoritmo evolutivo, como por ejemplo, la función de *fitness*, el tamaño de la población, el número máximo de generaciones, las tasas de recombinación y de mutación, etc.

La gramática independiente del contexto utilizada en la resolución del problema se define como $G = \{\Sigma_T, \Sigma_N, S, P\}$, donde el conjunto de terminales Σ_T es:

$$\Sigma_T = \{\sin, \cos, \exp, \log, +, -, *, x, (,)\}$$

el conjunto Σ_N de no terminales es el siguiente:

$$\Sigma_N = \{expr, pre\_op, var\}$$

y el axioma,

$$S = expr$$

Por último, el conjunto de producciones P contiene las siguientes reglas:

(0) $< expr >$ $\rightarrow$ $<expr> + <expr>$
(1) | $<expr> - <expr>$
(2) | $<expr> * <expr>$
(3) | $(<expr>)$
(4) | $<pre\_op> (<expr>)$
(5) | $<var>$

(0) $<pre\_op>$ $\rightarrow$ sin
(1) | cos
(2) | exp
(3) | log

(0) $<var>$ $\rightarrow$ x

En cuanto a la función de *fitness* se adoptaron las definiciones de [Koza (1992)]. El *raw fitness* se define como la medida de adecuación del individuo

expresada en términos propios del ámbito del problema que se está resolviendo. En particular, en el caso de regresión simbólica, el *raw fitness* es la suma del error en valor absoluto en 21 puntos de ajuste equiespaciados en el intervalo $[-1, +1]$, que son los puntos $x_0 = -1$, $x_1 = -0,9$, $x_2 = -0,8$, ..., $x_{20} = 1$. Con esta definición el mejor individuo es aquel que tiene el menor valor de *fitness*. Efectivamente, si un individuo ajusta perfectamente en los 21 puntos, su *raw fitness* será 0. Sin embargo, a priori, no se conoce el límite superior del *raw fitness*. Como en el proceso evolutivo se utiliza selección proporcional al *fitness* para elegir a los progenitores, es necesario que el mejor individuo sea aquel que tiene mayor *fitness* y por lo tanto, no se puede utilizar el *raw fitness*. En su lugar, se calcula el *fitness ajustado* de acuerdo con la definición de [Koza (1992)] en la que se establece que el *fitness ajustado* $a(i,t)$ de un individuo i en un instante de tiempo t se calcula a partir del *raw fitness* $r(i,t)$ con la siguiente fórmula:

$$a(i,t) = \frac{1}{1 + r(i,t)}$$

El *fitness ajustado* definido de esta manera se encuentra entre 0 y 1, y es mayor cuanto mejor sea el individuo.

Si ocurre que en la creación de un individuo se consumen todos los codones de su genotipo y su fenotipo contiene aún símbolos no terminales, se le asigna el peor valor de fitness, en este caso 0, y la presión selectiva se encargará de hacerlo desaparecer de la población.

Una vez diseñada la gramática y la función de fitness, se establecen los valores de los parámetros configurables del algoritmo como son el tamaño de la población, la longitud inicial del genotipo, etc. Todos estos parámetros se pueden variar a través de la interfaz proporcionada por la aplicación **GEEMMA** desarrollada como parte de este trabajo y cuya descripción detallada se puede leer en el capítulo 7.

Se realizaron muchas series de pruebas manteniendo fijos algunos parámetros y variando otros, buscando reproducir los valores de rendimiento mostrados en [O'Neill y Ryan (2003)] pero sin éxito. Por ello se tomaron las siguientes decisiones. En primer lugar, mantener fijos algunos parámetros con el único criterio de que coincidieran con los que aparecen en [O'Neill y Ryan (2003)]. Y en segundo lugar, hacer cuatro experimentos diferentes variando las tasas de recombinación y mutación, así como el uso/desuso de elitismo.

Los parámetros que se mantuvieron fijos se muestran en la tabla 5.2.

De los siete parámetros de la tabla 5.2, el número máximo de generaciones se aumentó de 50 (según aparece en [O'Neill y Ryan (2003)]) a 100 únicamente para ver si en el rango de 51 a 100 generaciones aumentaba bruscamente el

Tamaño de la población	500
Número máximo de generaciones	100
Número mínimo de codones (inicial)	1
Número máximo de codones (inicial)	10
Valor mínimo de codón	0
Valor máximo de codón	256
Operador de *wrapping*	inhabilitado

Tabla 5.2: Parámetros de ejecución del algoritmo EG para el problema de regresión simbólica.

rendimiento que no se había obtenido en el rango de 1 a 50. Aunque una vez que se aclaró la confusión en las gráficas de rendimiento, según se explica en el capítulo 4, se observa que bastaría con que el número máximo de generaciones fuera 50.

Los parámetros que se variaron y que originaron distintos experimentos fueron los siguientes:

- **Uso de elistismo**: se hicieron pruebas utilizando elitismo y sin él. Aunque en el capítulo dedicado a **GEEMMA** se puede encontrar una descripción detallada del algoritmo de EG ajustada y de los parámetros del mismo, en este punto se recuerda que el algoritmo evolutivo utilizado aplica una técnica de reemplazo generacional combinada con el uso o no de elitismo. Si se activa el uso de elitismo, se mantiene el mejor individuo de una generación a la siguiente, y no se mantiene en el caso de que esté desactivado su uso.

- **Tasa de recombinación (%)**: se hicieron pruebas con tasas de recombinación del 90 % y el 100 %.

- **Tasa de mutación (%)**: se probaron tasas de mutación desde el 0 % hasta el 100 % en incrementos de 10 puntos.

Se puede encontrar una explicación completa del significado de los parámetros en el capítulo dedicado a la descripción de **GEEMMA** .

En resumen, se hicieron cuatro experimentos, todos ellos con la misma gramática, la misma función de *fitness* y los mismos valores de los parámetros fijos que se han descrito anteriormente (tamaño de la población, número máximo de generaciones, número mínimo y máximo de codones, valor mínimo y máximo de codón y operador de wrapping deshabilitado). En cada experimento se fijó el uso o no de elitismo, la tasa de recombinación (90 % o 100 %) y se hicieron 11 series de 100 ejecuciones independientes para tasas de

mutación desde 0 % hasta 100 % con incrementos de 10 puntos. En la tabla 5.3 se muestra un esquema de los cuatro experimentos realizados.

Experimento	Tasa recombinación(%)	Elitismo	Tasa de mutación(%)
1	90	NO	0-10-20...100
2	90	SI	0-10-20...100
3	100	NO	0-10-20...100
4	100	SI	0-10-20...100

Tabla 5.3: Grupos de experimentos para resolver el problema de regresión simbólica con EG.

En el primer experimento se utilizó una tasa de recombinación del 90 %, se desactivó el elitismo y la tasa de mutación se varió desde el 0 % hasta el 100 % en incrementos de 10 puntos. En la figura 5.15 se muestra la probabilidad acumulada de éxito en % en función del número de generación para cada una de las 11 series de 100 ejecuciones independientes correspondientes a las 11 tasas de mutación diferentes. Se puede observar que los mejores rendimientos se obtienen para tasas de mutación del 50 % y 60 % representadas en colores amarillo y naranja respectivamente. En concreto, para la tasa de mutación del 50 % se alcanza una probabilidad acumulada de éxito del 68 % en la generación 50 y del 96 % en la generación 92. En la figura también se observa la gran dispersión en los rendimientos obtenidos para las distintas tasas de mutación. Por ejemplo, en la generación 50, las probabilidades acumuladas de éxito oscilan dentro del intervalo [21,68] con una amplitud de 47 puntos. Dicha amplitud de variación de la probabilidad acumulada de éxito para distintas tasas de mutación de aproximadamente 50 puntos, se mantiene a partir de la generación 50 hasta la 100.

En el segundo experimento se mantuvo la tasa de recombinación en 90 % pero se habilitó el uso de elitismo. Los resultados se muestran en la figura 5.16. Se puede apreciar que se minimiza sustancialmente la dispersión de las series correspondientes a distintas tasas de mutación. En la generación 50, las probabilidades acumuladas de éxito que oscilaban en el primer experimento dentro del intervalo [21,68] con una amplitud de 47 puntos, pasan a

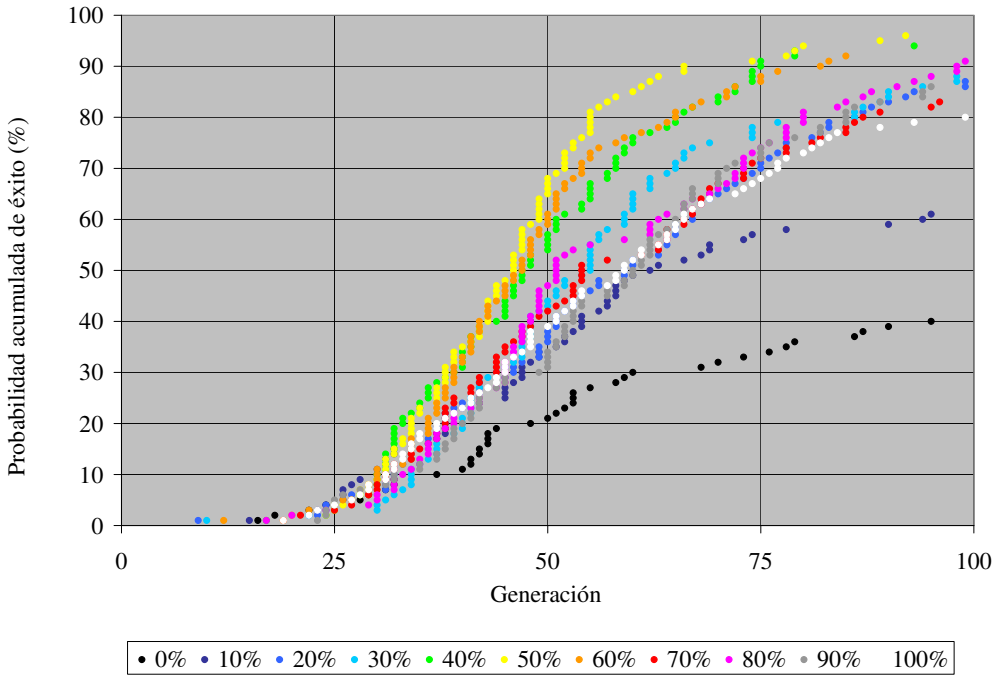

Figura 5.15: Probabilidad acumulada de éxito de EG para el problema de regresión simbólica, calculada sobre 11 series de 100 ejecuciones independientes para 11 tasas diferentes de mutación, una tasa de recombinación del 90 % y sin uso de elitismo.

oscilar al intervalo más reducido [39,70] que tiene una amplitud de 31 puntos. En la generación 75, el intervalo en el que oscilaban las probabilidades acumuladas de éxito en el primer experimento era [33,91] con una amplitud de 58 puntos, se reduce en el segundo experimento a [63,91] con una amplitud de 28 puntos. Junto con la reducción de la dispersión, ha mejorado el rendimiento de las series correspondientes a la tasas de mutación más bajas, situación explicable por la conservación del mejor individuo de generación en generación que proporciona el elitismo. Por otra parte se observa que el mejor rendimiento sigue siendo para la tasa de mutación del 50 %.

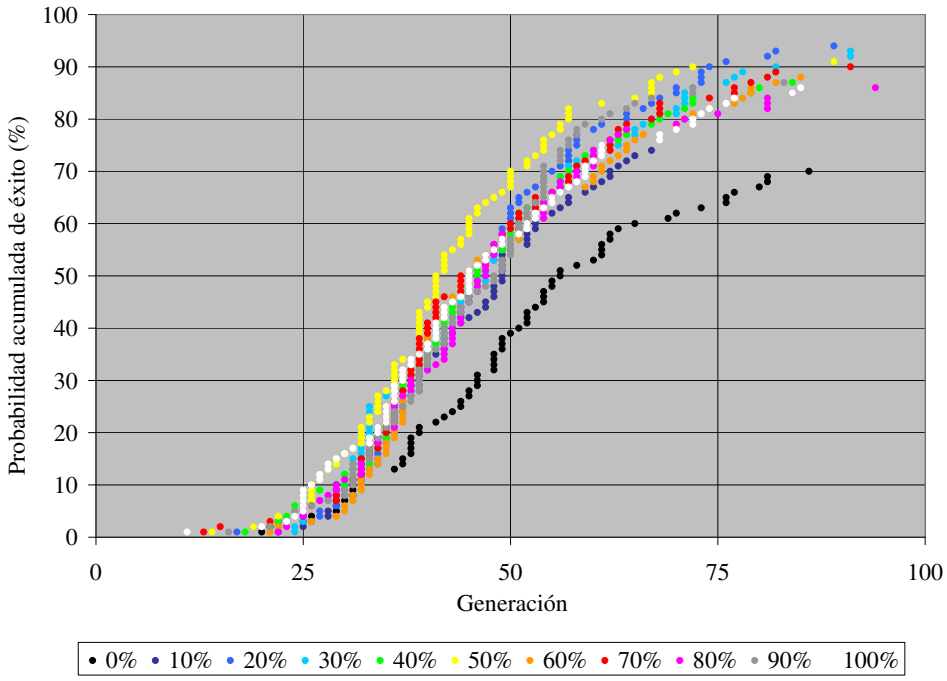

Figura 5.16: Probabilidad acumulada de éxito de EG para el problema de regresión simbólica, calculada sobre 11 series de 100 ejecuciones independientes para 11 tasas diferentes de mutación, una tasa de recombinación del 90 % y uso de elitismo.

En el tercer experimento se aumentó la tasa de recombinación al 100 %, se desactivó el elitismo, y de nuevo se utilizaron tasas de mutación desde el 10 % hasta el 100 % con incrementos de 10 puntos. En la figura 5.17 se muestran las series de probabilidad acumulada de éxito para las distintas tasas de mutación. Se puede observar que se mantiene la dispersión de las series que ocurría en el primer experimento en el que tampoco se utilizaba elitismo y la tasa de recombinación era inferior (90 %). En las gráficas de rendimiento se puede observar que en el tercer experimento las series correspondientes a tasas de mutación de 0 %, 10 %, 20 % y 30 % se mantienen dispersas pero el resto de las series se agrupan y además con probabilidades acumuladas de éxito superiores a las obtenidas en el primer experimento. Este fenómeno de mejora del rendimiento con el incremento de la tasa de recombinación parece esperable ya que se supone que el proceso de recombinación puede generar individuos mejores y por lo tanto evolucionar más rápidamente hacia una solución.

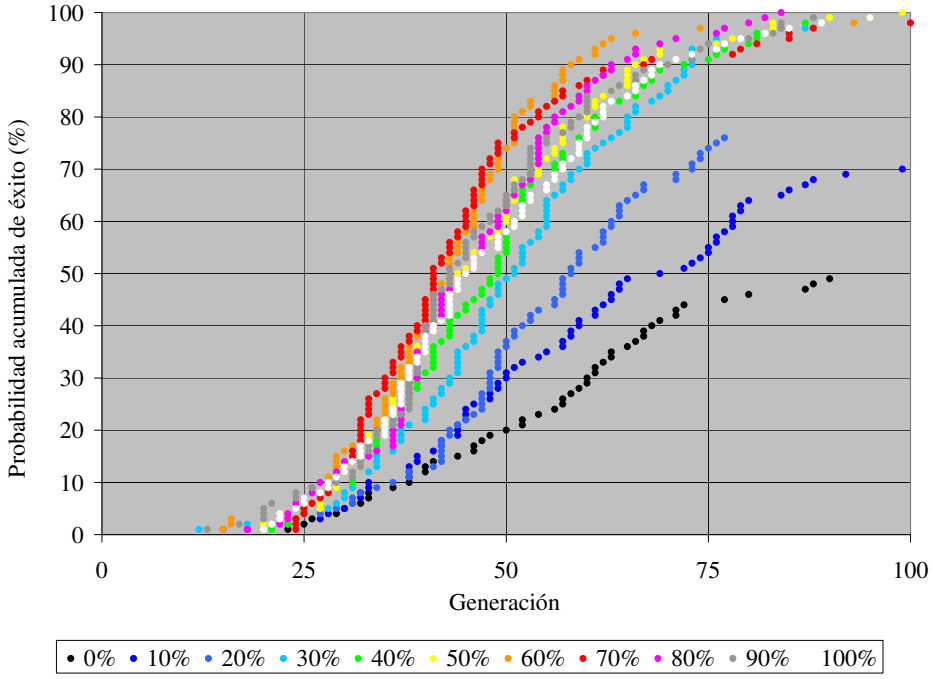

Figura 5.17: Probabilidad acumulada de éxito de EG para el problema de regresión simbólica, calculada sobre 11 series de 100 ejecuciones independientes para 11 tasas diferentes de mutación, una tasa de recombinación del 100 % y sin uso de elitismo.

Por último, el cuarto experimento se realizó con uso de elitismo y una tasa de recombinación del 100 %. De nuevo se observa que disminuye la dispersión de las series si se compara con el tercer experimento en el que también la tasa de recombinación era del 100 % pero no se utilizaba elitismo. La tasa de mutación del 100 % proporciona un mejor rendimiento hasta la generación 50, y a partir de dicha generación es la tasa de mutación del 60 % la que tiene una frecuencia acumulada de éxito superior. En la figura 5.18 se muestra la probabilidad acumulada de éxito en % para distintas tasas de mutación y 100 ejecuciones independientes para cada tasa.

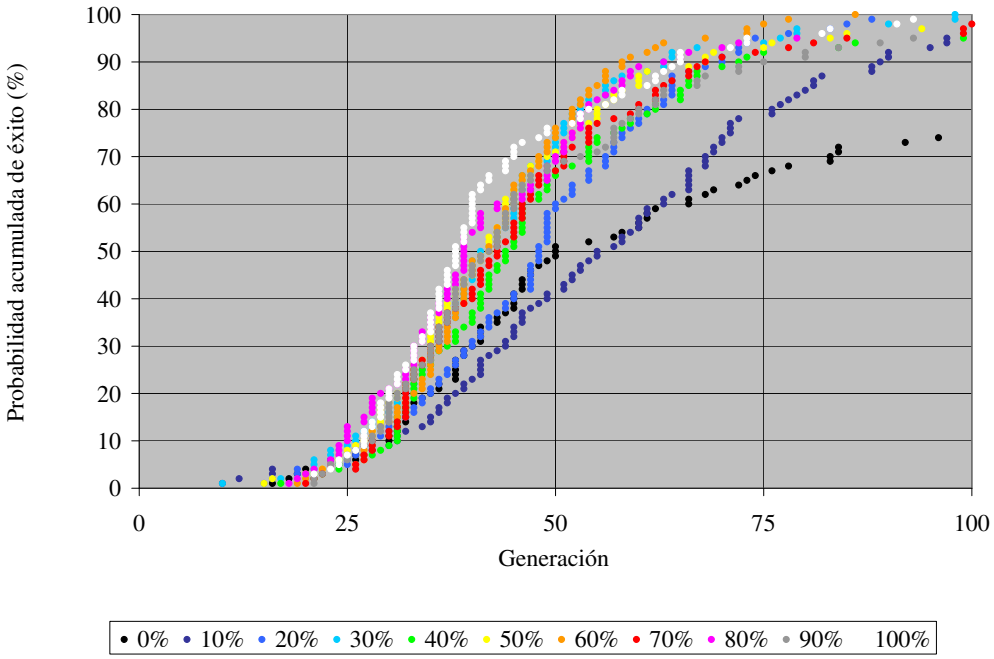

Figura 5.18: Probabilidad acumulada de éxito de EG para el problema de regresión simbólica, calculada sobre 11 series de 100 ejecuciones independientes para 11 tasas diferentes de mutación, una tasa de recombinación del 100 % y uso de elitismo.

5.3.3. Solución del problema con EGA

La definición de la semántica es uno de los aspectos más importantes en el planteamiento de la solución a un problema con EGA ya que, mediante la semántica se define la "calidad" de los individuos de la población, y la hipótesis de partida es que al aumentar la calidad de los individuos se incrementa la calidad de la población y como consecuencia de ello se mejora el rendimiento para encontrar soluciones ya que, de alguna manera se restringe el espacio de búsqueda.

Es obvio que, para un mismo problema, se pueden plantear distintas semánticas. En concreto, para el problema de regresión simbólica se han propuesto dos semánticas que han conducido a los dos experimentos que se describen a continuación.

5.3.3.1. Primer experimento

En el primer experimento la semántica diseñada tiene como objetivo rechazar aquellos individuos que contengan en su fenotipo subexpresiones que no estén definidas en alguno de los 21 puntos de evaluación de la función. En la gramática independiente del contexto que se ha utilizado para resolver el problema con EG, ésta condición de rechazo sólo es cierta cuando aparece el logaritmo de una expresión cuyo valor es menor o igual que 0. Si durante la construcción de un fenotipo, se detecta esta circunstancia, se interrumpe el proceso de generación del fenotipo, y el individuo es rechazado, es decir, no entra a formar parte de la población.

Informalmente, valdría decir que los atributos necesarios para incorporar este contenido semántico son los siguientes:

- 21 atributos de tipo real asociados al no terminal <*expr*>. Cada atributo almacena la evaluación de la subexpresión representada por <*expr*> en uno de los 21 puntos de evaluación.
- 21 atributos de tipo real definidos para el no terminal <*var*>. Cada atributo almacena uno de los 21 puntos de evaluación.
- un atributo de tipo cadena de caracteres para el símbolo <*pre_op*> que almacena la función concreta ("sin", "cos", "exp", "log").

También de manera informal, el conjunto de acciones semánticas tiene como objetivo evaluar durante la construcción del árbol de análisis las subexpresiones en los 21 puntos de control, lo que permite descartar lo antes posible los individuos que contengan subexpresiones que no estén definidas en alguno de los 21 puntos.

Es evidente que la semántica expresada por la gramática de atributos que se ha descrito informalmente se podría trasladar a la función de *fitness* de EG. Efectivamente, en la función de *fitness* se puede comprobar si el fenotipo contiene subexpresiones que no estén definidas en alguno de los 21 puntos de control, y si ocurre tal circunstancia, penalizar de alguna manera al individuo en cuestión. Por lo tanto, surge la pregunta ¿si la semántica se puede trasladar a la función de *fitness* por qué plantearse EGA?. La respuesta tiene dos vertientes, una relacionada con el tiempo de proceso y otra con la capacidad expresiva (confort de diseño). En cuanto al tiempo de proceso, si la semántica se traslada al fitness, como los fenotipos se construyen completamente antes de ser evaluados, no se puede "ahorrar" el

tiempo que sí se ahorra en EGA al interrumpir la construcción del fenotipo en el momento en el que se detecte una subexpresión no definida en alguno de los 21 puntos de evaluación. En relación a la capacidad expresiva, es muy sencillo, directo y confortable expresar con una gramática de atributos la condición de subexpresión no definida en algún punto ya que, simplemente implica evaluar el argumento de la función logaritmo para detectar si es negativo. Sin embargo, en la función *fitness*, para implementar un control similar sería necesario diseñar un algoritmo que buscara dentro del fenotipo las funciones logaritmo, siendo esto más complicado y confuso.

La gramática de atributos que previamente se definió de manera informal, formalmente se define como $G_1 = \{\Sigma_T, \Sigma_N, S, P, K\}$, donde el conjunto de terminales Σ_T es:

$$\Sigma_T = \{\sin, \cos, \exp, \log, +, -, *, x, (,)\}$$

Ninguno de los símbolos terminales tiene atributos definidos y por este motivo, para simplificar la notación, se omiten los paréntesis que acompañan a cada símbolo en la definición formal de una gramática de atributos.

El conjunto de los símbolos no terminales Σ_N es define como:

$$\begin{aligned}\Sigma_N = \{ \quad & expr\ (\mathbb{R} \quad valor_i\ ,,\ i = 0, 1, \ldots, 20), \\ & var\ (\mathbb{R} \quad punto_i\ ,,\ i = 0, 1, \ldots, 20), \\ & pre\_op\ ([\ \text{“sin”},\ \text{“cos”},\ \text{“exp”},\ \text{“log”}] \quad funcion) \quad \}\end{aligned}$$

donde el atributo $valor_i$ almacena el valor de la subexpresión representada por el no terminal <*expr*> en el punto i, el atributo $punto_i$ guarda el valor del punto de evaluación i, y por último, el atributo *funcion* almacena una cadena de caracteres que define la función representada por el símbolo <*pre_op*>.

El axioma de la gramática de atributos es:

$$S = expr$$

Por último, no se define información global de la gramática de atributos, $K = \Phi$.

El conjunto P de producciones junto con sus acciones semánticas es el siguiente:

(0) $<expr> \rightarrow <expr>_1 + <expr>_2$
{
DESDE $i = 0$ **HASTA** 20
$<expr>.valor_i = <expr>_1.valor_i + <expr>_2.valor_i$
}

(1) $<expr> \rightarrow <expr>_1 - <expr>_2$
{
DESDE $i = 0$ **HASTA** 20
$<expr>.valor_i = <expr>_1.valor_i - <expr>_2.valor_i$
}

(2) $<expr> \rightarrow <expr>_1 * <expr>_2$
{
DESDE $i = 0$ **HASTA** 20
$<expr>.valor_i = <expr>_1.valor_i * <expr>_2.valor_i$
}

(3) $<expr> \rightarrow (<expr>_1)$
{
DESDE $i = 0$ **HASTA** 20
$<expr>.valor_i = <expr>_1.valor_i$
}

(4) $<expr> \rightarrow <pre\_op> (<expr>_1)$
{
DESDE $i = 0$ **HASTA** 20
SI ($<pre\_op>.funcion =$ "log") **Y** ($<expr>_1.valor_i \leq 0$)
ERROR SEMÁNTICO
SI NO $<expr>.valor_i =$ **APLICAR** $<pre\_op>.funcion$ **A** $<expr>_1.valor_i$
} [1]

(5) $<expr> \rightarrow (<var>)$
{
DESDE $i = 0$ **HASTA** 20
$<expr>.valor_i = <var>.punto_i$
}

(0) $<pre\_op> \rightarrow$ sin { $<pre\_op>.funcion =$ "sin" }
(1) $<pre\_op> \rightarrow$ cos { $<pre\_op>.funcion =$ "cos" }
(2) $<pre\_op> \rightarrow$ cos { $<pre\_op>.funcion =$ "exp" }
(3) $<pre\_op> \rightarrow$ cos { $<pre\_op>.funcion =$ "log" }

[1] El pseudocódigo $<expr>.valor_i =$ **APLICAR** $<pre\_op>.funcion$ **A** $<expr>_1.valor_i$ quiere decir que el valor del atributo $valor_i$ del no terminal $<expr>$ se calcula aplicando la función que indica el atributo *funcion* del no terminal $<pre\_op>$ al valor del atributo $valor_i$ del símbolo $<expr>_1$

$$
\begin{array}{llll}
(0) & <var> & \rightarrow & x \quad \{ \quad <var>.punto_0 = -1{,}0 \\
 & & & \qquad <var>.punto_1 = -0{,}9 \\
 & & & \qquad <var>.punto_2 = -0{,}8 \\
 & & & \qquad <var>.punto_3 = -0{,}7 \\
 & & & \qquad <var>.punto_4 = -0{,}6 \\
 & & & \qquad <var>.punto_5 = -0{,}5 \\
 & & & \qquad <var>.punto_6 = -0{,}4 \\
 & & & \qquad <var>.punto_7 = -0{,}3 \\
 & & & \qquad <var>.punto_8 = -0{,}2 \\
 & & & \qquad <var>.punto_9 = -0{,}1 \\
 & & & \qquad <var>.punto_{10} = 0{,}0 \\
 & & & \qquad <var>.punto_{11} = 0{,}1 \\
 & & & \qquad <var>.punto_{12} = 0{,}2 \\
 & & & \qquad <var>.punto_{13} = 0{,}3 \\
 & & & \qquad <var>.punto_{14} = 0{,}4 \\
 & & & \qquad <var>.punto_{15} = 0{,}5 \\
 & & & \qquad <var>.punto_{16} = 0{,}6 \\
 & & & \qquad <var>.punto_{17} = 0{,}7 \\
 & & & \qquad <var>.punto_{18} = 0{,}8 \\
 & & & \qquad <var>.punto_{19} = 0{,}9 \\
 & & & \qquad <var>.punto_{20} = 1{,}0 \\
 & & & \}
\end{array}
$$

Utilizando la gramática G_1 se han realizado un conjunto de pruebas con el objetivo de poder comparar los resultados obtenidos, en términos del número de generaciones necesarias para encontrar solución, con los obtenidos para el mismo problema utilizando EG. Se han realizado 11 pruebas de 100 ejecuciones independientes cada una de ellas. Los valores de los parámetros configurables del algoritmo son iguales a los utilizados en EG para poder comparar, y se resumen en la tabla 5.4.

En las figuras 5.19 - 5.29 se muestran los rendimientos obtenidos en las 11 pruebas, comparados con los rendimientos obtenidos para el mismo problema resuelto con EG. En las gráficas se puede observar que en 7 de las 11 pruebas (en concreto, para las tasas de mutación 0 %, 10 %, 20 %, 70 %, 80 %, 90 % y 100 %) se mejora el rendimiento de manera considerable, llegando incluso a mejoras de más de 30 puntos como es el caso correspondiente a la tasa de mutación de 90 %. Por otra parte, en las pruebas correspondientes a las tasas de mutación de 30 %, 40 %, 50 % y 60 %, el rendimiento obtenido es similar.

La mejora obtenida en el rendimiento de EGA con respecto a EG para el problema de regresión simbólica no se alcanzó en experimentos realizados anteriormente y cuyos resultados se pueden consultar en [de˜la Cruz y otros

Tamaño de la población	500
Número máximo de generaciones	100
Número mínimo de codones (inicial)	1
Número máximo de codones (inicial)	10
Valor mínimo de codón	0
Valor máximo de codón	256
Operador de *wrapping*	inhabilitado
Elitismo	NO
Tasa de recombinación	90 %
Tasa de mutación	0 %, 10 %, 20 %, ..., 100 %

Tabla 5.4: Parámetros de ejecución del primer experimento del algoritmo EGA para el problema de regresión simbólica.

(2005) y Ortega y otros (2007)]. Esta mejora es debida a la corrección de algunos errores detectados en **GEEMMA** .

En cuanto al número de individuos rechazados por incumplimiento de la restricción semántica establecida, se ha realizado el siguiente cálculo. En cada ejecución independiente que termina con éxito (encontrando una solución) se registra el número de individuos rechazados. Este valor, se divide por la generación en la que ha terminado la ejecución, y se supone, que se obtiene una estimación del promedio de individuos rechazados por generación. Lo anterior es una suposición y no una afirmación porque no se ha realizado, para cada ejecución independiente, ningún estudio acerca de la evolución de individuos rechazados a medida que avanza el proceso evolutivo hacia la solución. Este estudio se deja abierto para futuras investigaciones. A partir del promedio de individuos descartados por generación en una ejecución independiente, se calcula para cada tasa de mutación, la media y la desviación estándar de ese parámetro (se recuerda que se hacen 100 ejecuciones independientes para cada tasa de mutación).

En la tabla 5.5 se muestran los valores obtenidos en este primer experimento. Se puede observar que para los 11 casos de mutación, la media y la desviación estándar del número de individuos rechazados en cada generación, son similares en orden de magnitud. Si se calcula la media de las medias de los 11 casos, se podría decir que en el problema de regresión simbólica resuelto con EGA, con los parámetros de ejecución descritos en su momento (gramática de atributos, función de *fitness*, tamaño de la población, etc) en promedio, se rechazan 21 individuos en cada generación, que equivale al 4,2 %

de la población.

Tasa de mutación	media	desviación estándar
0 %	10	8
10 %	10	6
20 %	12	6
30 %	15	6
40 %	18	6
50 %	21	6
60 %	24	5
70 %	26	5
80 %	30	5
90 %	32	6
100 %	36	6

Tabla 5.5: Media y desviación estándar del número de individuos rechazados en cada generación del primer experimento de EGA para el problema de regresión simbólica.

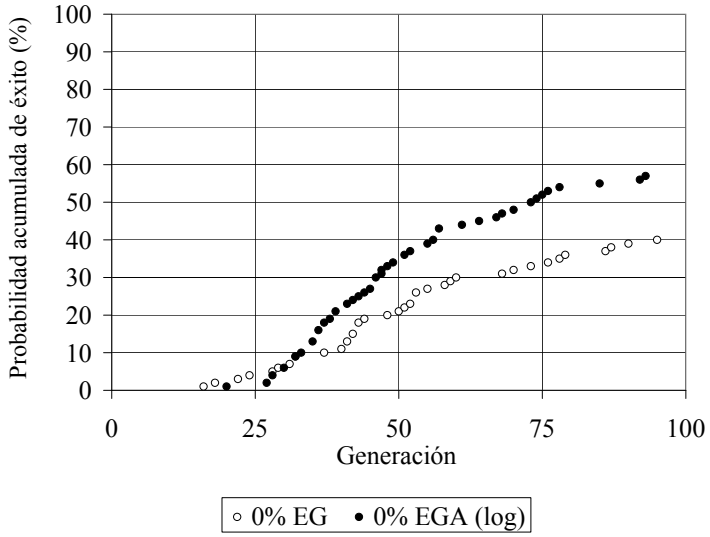

Figura 5.19: Comparativa de las probabilidades acumuladas de éxito de EG y EGA para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 0 %, una tasa de recombinación del 90 % y sin uso de elitismo.

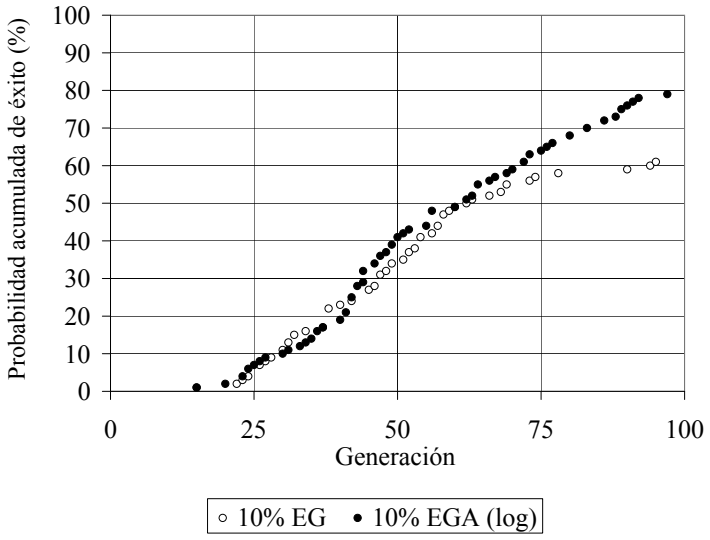

Figura 5.20: Comparativa de las probabilidades acumuladas de éxito de EG y EGA para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 10 %, una tasa de recombinación del 90 % y sin uso de elitismo.

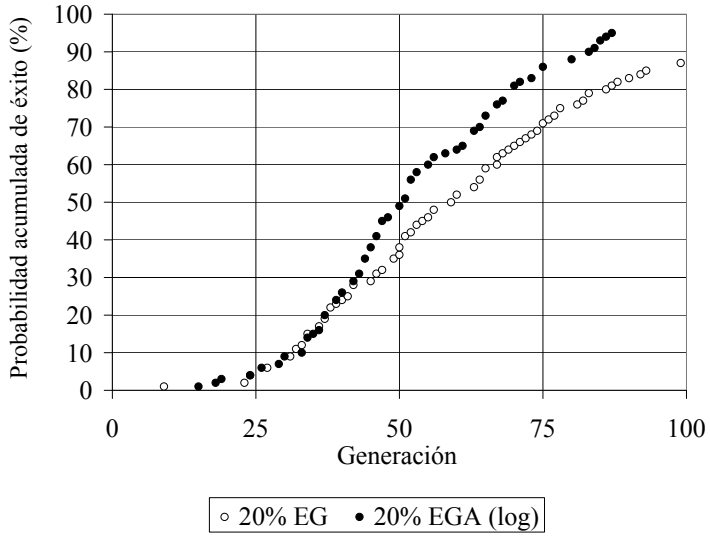

Figura 5.21: Comparativa de las probabilidades acumuladas de éxito de EG y EGA para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 20 %, una tasa de recombinación del 90 % y sin uso de elitismo.

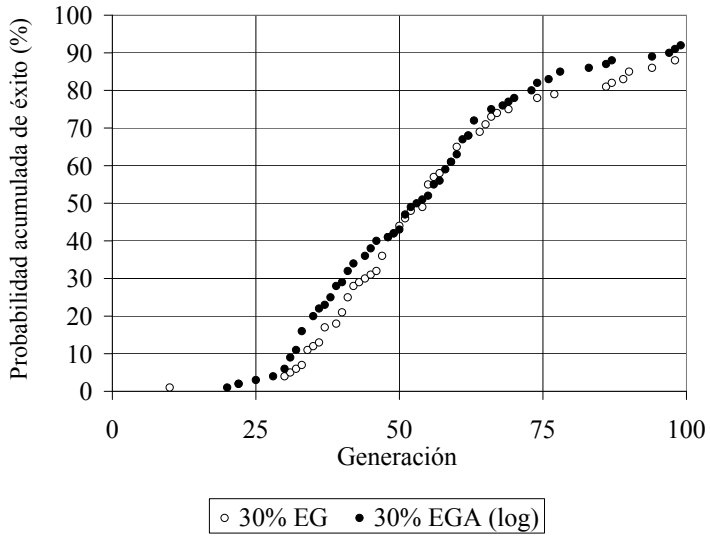

Figura 5.22: Comparativa de las probabilidades acumuladas de éxito de EG y EGA para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 30 %, una tasa de recombinación del 90 % y sin uso de elitismo.

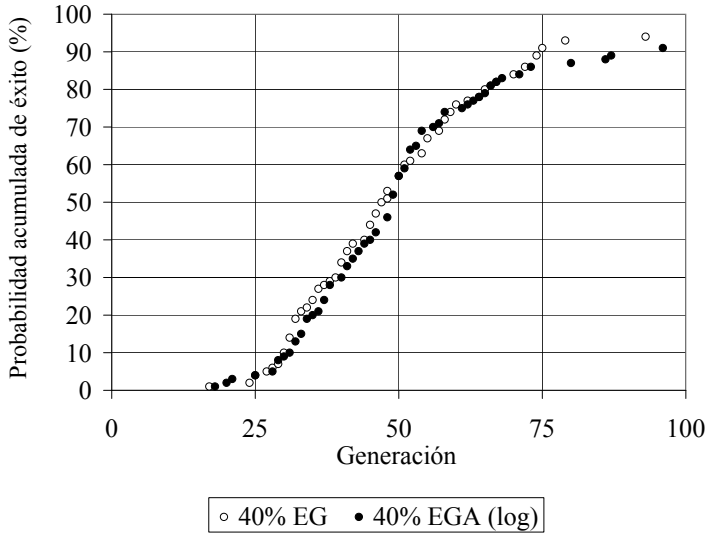

Figura 5.23: Comparativa de las probabilidades acumuladas de éxito de EG y EGA para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 40 %, una tasa de recombinación del 90 % y sin uso de elitismo.

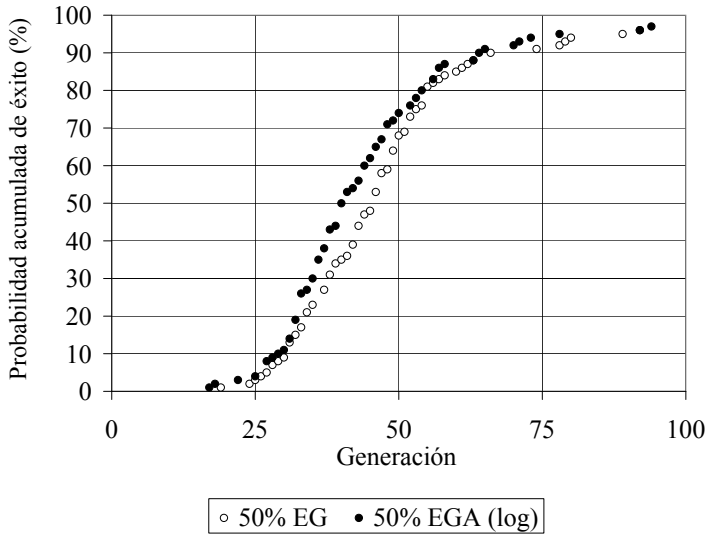

Figura 5.24: Comparativa de las probabilidades acumuladas de éxito de EG y EGA para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 50 %, una tasa de recombinación del 90 % y sin uso de elitismo.

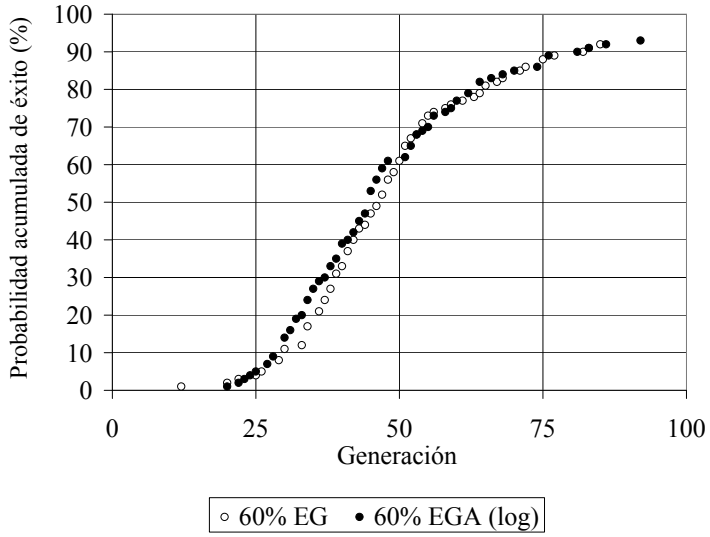

Figura 5.25: Comparativa de las probabilidades acumuladas de éxito de EG y EGA para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 60 %, una tasa de recombinación del 90 % y sin uso de elitismo.

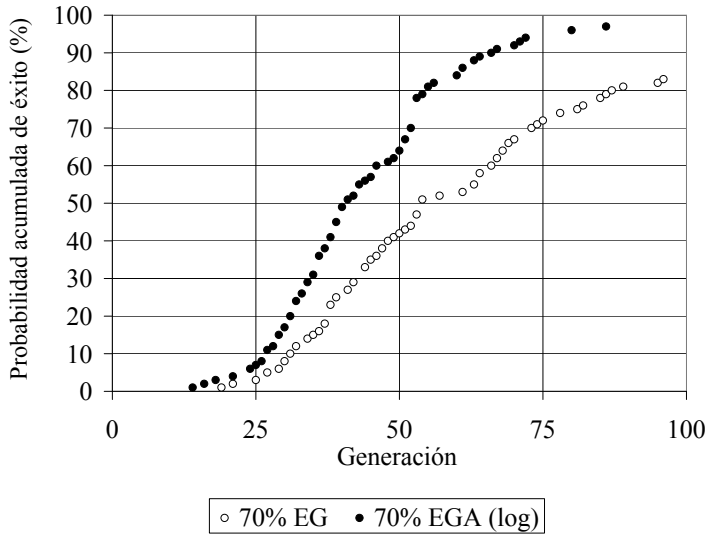

Figura 5.26: Comparativa de las probabilidades acumuladas de éxito de EG y EGA para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 70 %, una tasa de recombinación del 90 % y sin uso de elitismo.

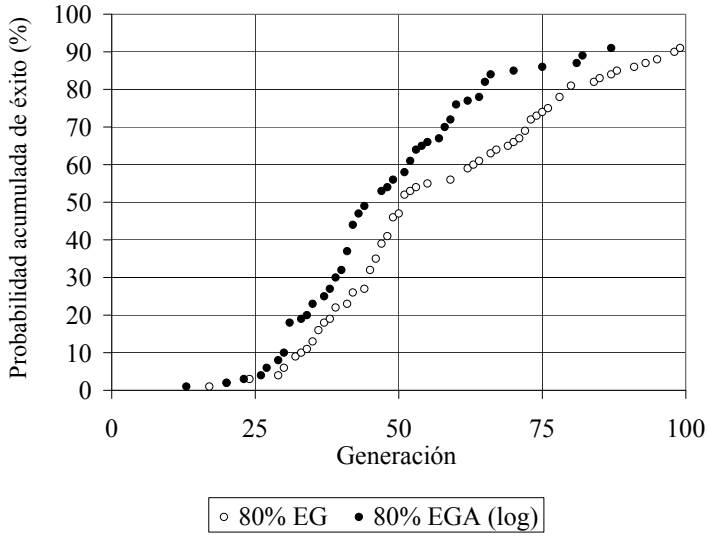

Figura 5.27: Comparativa de las probabilidades acumuladas de éxito de EG y EGA para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 80 %, una tasa de recombinación del 90 % y sin uso de elitismo.

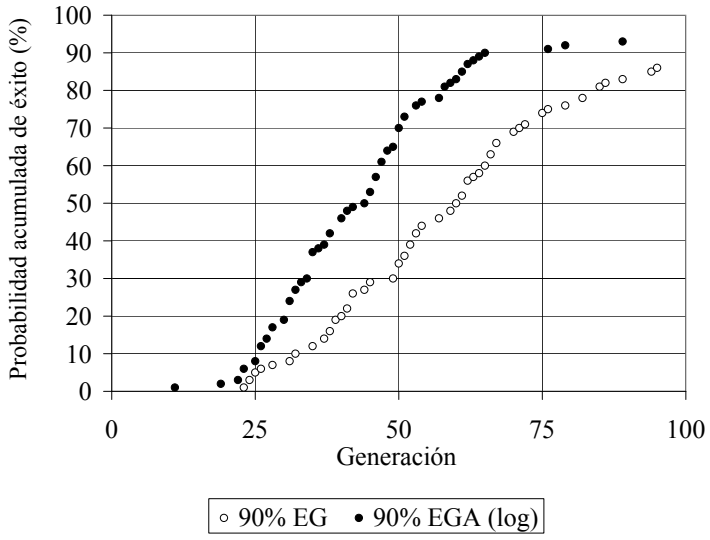

Figura 5.28: Comparativa de las probabilidades acumuladas de éxito de EG y EGA para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 90 %, una tasa de recombinación del 90 % y sin uso de elitismo.

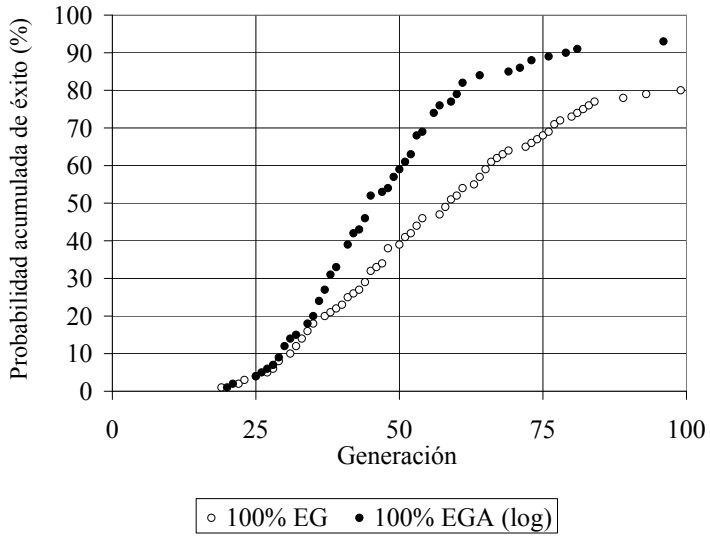

Figura 5.29: Comparativa de las probabilidades acumuladas de éxito de EG y EGA para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 100 %, una tasa de recombinación del 90 % y sin uso de elitismo.

5.3.3.2. Segundo experimento

En el segundo experimento se "endurece" la semántica de manera que a los individuos que ya se rechazaban en el primer experimento, se añaden también aquellos que no ajusten de manera exacta la función objetivo en al menos alguno los puntos -1, 0 ó 1. La gramática de atributos se define a partir de la diseñada para el primer experimento, manteniendo el sistema de atributos e incorporando en las acciones semánticas del axioma la nueva restricción, es decir, la comprobación del ajuste exacto en alguno de los puntos -1, 0 ó 1. Evidentemente, esta nueva restricción sólo se puede comprobar una vez terminada la generación del fenotipo, a diferencia del primer experimento en el que las restricciones semánticas son comprobables durante la construcción del mismo.

Formalmente se describe la gramática utilizada en este experimento como $G_2 = \{\Sigma_T, \Sigma_N, S, P, K\}$, donde los conjuntos Σ_T, Σ_N, K y el axioma S coinciden con los de la gramática G_1, y el conjunto P de producciones se obtiene modificando el conjunto P de la gramática G_1 de manera que en todas las reglas del axioma se incorpora la comprobación del ajuste exacto en alguno de los puntos -1, 0 ó 1. Como ejemplo, se muestra a continuación la acción semántica de la primera regla del axioma.

(0) <*expr*> → <*expr*>$_1$ + <*expr*>$_2$
{
DESDE $i = 0$ **HASTA** 20
<*expr*> .*valor*$_i$ =<*expr*>$_1$.*valor*$_i$+ <*expr*>$_2$.*valor*$_i$
SI (<*exp*> ES LA RAÍZ DEL ÁRBOL DE DERIVACIÓN) **Y**
(<*expr*> .*valor*$_0$! = 0) Y (<*expr*> .*valor*$_{10}$! = 0) Y (<*expr*> .*valor*$_{20}$! = 4)
ERROR SEMÁNTICO
}

Se han realizado de nuevo 11 pruebas similares a las realizadas en el primer experimento y sus resultados junto con los obtenidos con la gramática G_1 se muestran en las figuras 5.30 - 5.40. Se puede observar que en 8 de los 11 casos, el rendimiento obtenido en el segundo experimento, con la semántica más restrictiva, es superior al obtenido en el primero, con la semántica más relajada. Estos casos de mejora corresponden a las tasas de mutación de 10 %, 30 %, 40 %, 50 %, 60 %, 80 %, 90 % y 100 %. Por otra parte, el rendimiento se mantiene similar en los casos correspondientes a las tasas de mutación de 0 % y 70 %. Por último, se puede observar que solamente hay un caso, el correspondiente a una tasa de mutación del 20 %, en el que el rendimiento obtenido con la semántica más restrictiva es inferior al obtenido con la semántica más relajada.

En cuanto al número de individuos rechazados, se realizan los mismos cálculos que en el primer experimento. Los resultados se muestran en la tabla 5.6. Se observa que, de nuevo, tanto la media como la desviación estándar se mantienen parecidas para las distintas tasas de mutación. Si se calcula el promedio de las medias, se podría decir que en este experimento se rechazan de media 39 individuos en cada generación, que equivale al 7,8 % de la población.

En este segundo experimento, el promedio del número de individuos rechazados en cada generación es superior al obtenido en el primer experimento. Parece un resultado razonable, ya que, con una semántica más restrictiva, es lógico que haya más individuos que no la satisfagan.

En resumen, en todas las pruebas realizadas, el rendimiento obtenido con EGA es superior al obtenido con EG, en algunos casos con la semántica definida con la gramática de atributos G_1 y en otros casos con la semántica definida con G_2.

Después de la realización de los dos experimentos presentados, se realizaron algunas pruebas individuales con distintos grados de semántica, observándose que si se endurece la semántica en exceso, el algoritmo no prospera. En ocasiones se debe a que la inicialización aleatoria de la población no

Tasa de mutación	media	desviación estándar
0 %	15	13
10 %	25	18
20 %	25	17
30 %	29	13
40 %	34	11
50 %	38	11
60 %	44	12
70 %	50	11
80 %	53	9
90 %	58	9
100 %	61	9

Tabla 5.6: Media y desviación estándar del número de individuos rechazados en cada generación del segundo experimento de EGA del problema de regresión simbólica.

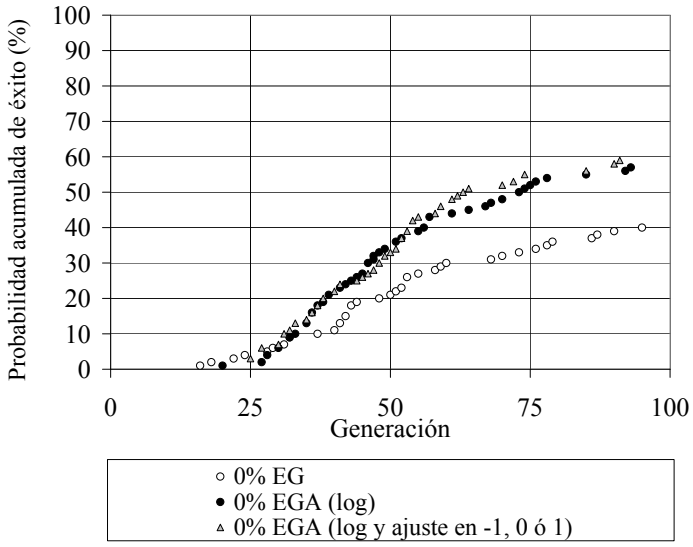

Figura 5.30: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 0 %, una tasa de recombinación del 90 % y sin uso de elitismo.

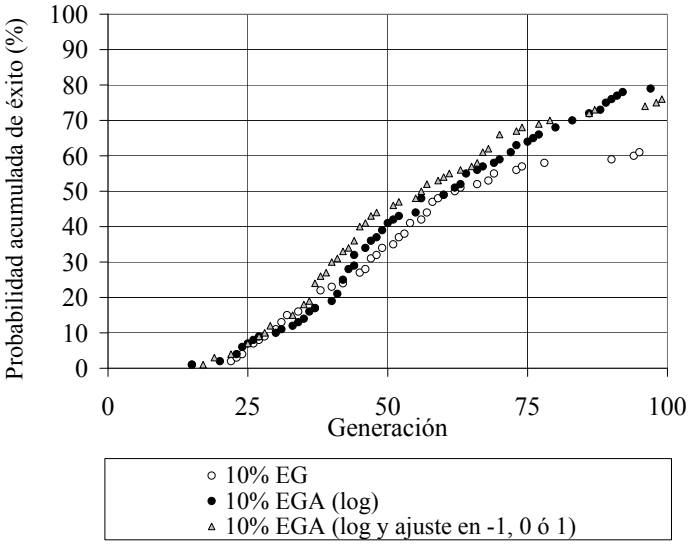

Figura 5.31: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 10 %, una tasa de recombinación del 90 % y sin uso de elitismo.

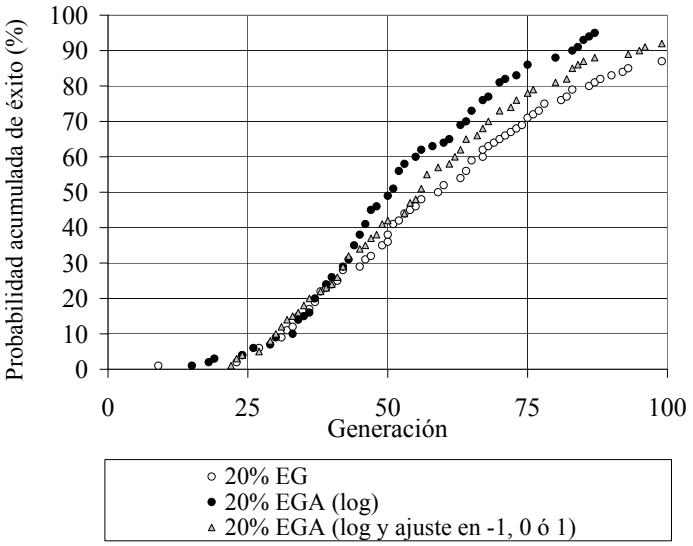

Figura 5.32: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 20 %, una tasa de recombinación del 90 % y sin uso de elitismo.

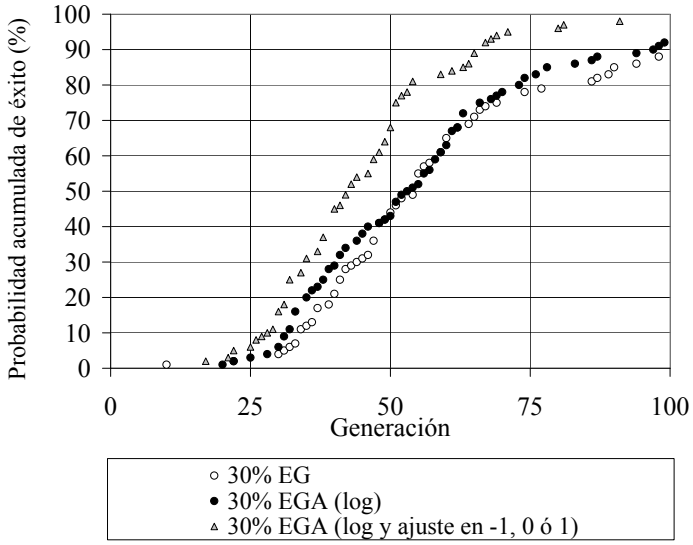

Figura 5.33: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 30 %, una tasa de recombinación del 90 % y sin uso de elitismo.

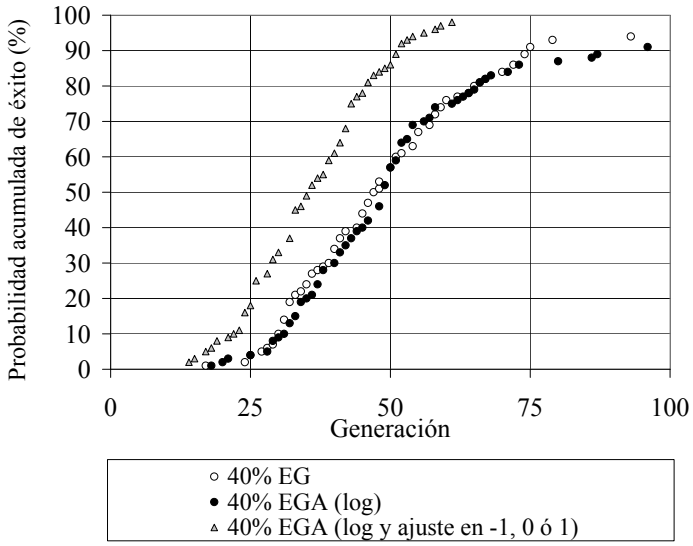

Figura 5.34: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 40 %, una tasa de recombinación del 90 % y sin uso de elitismo.

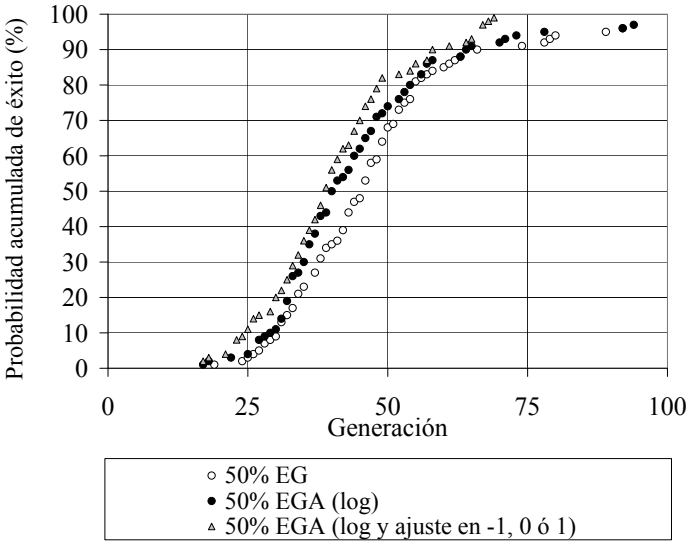

Figura 5.35: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 50 %, una tasa de recombinación del 90 % y sin uso de elitismo.

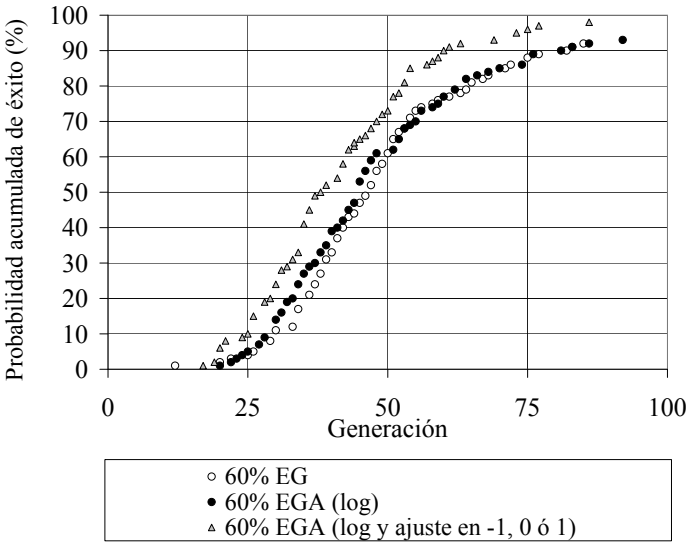

Figura 5.36: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 60 %, una tasa de recombinación del 90 % y sin uso de elitismo.

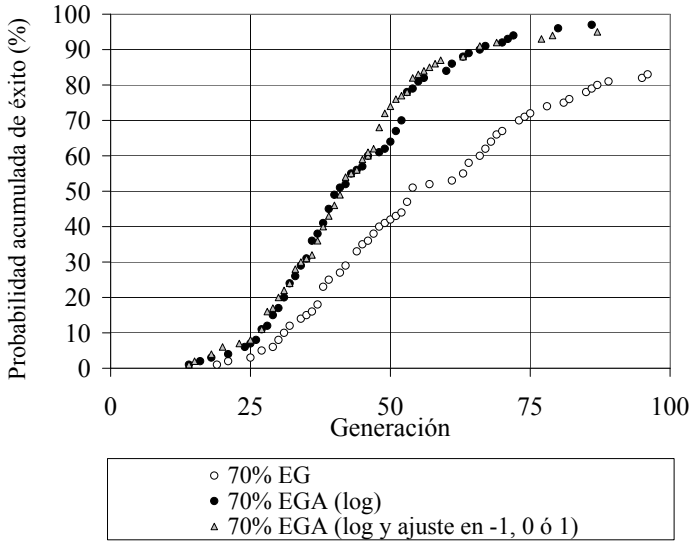

Figura 5.37: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 70 %, una tasa de recombinación del 90 % y sin uso de elitismo.

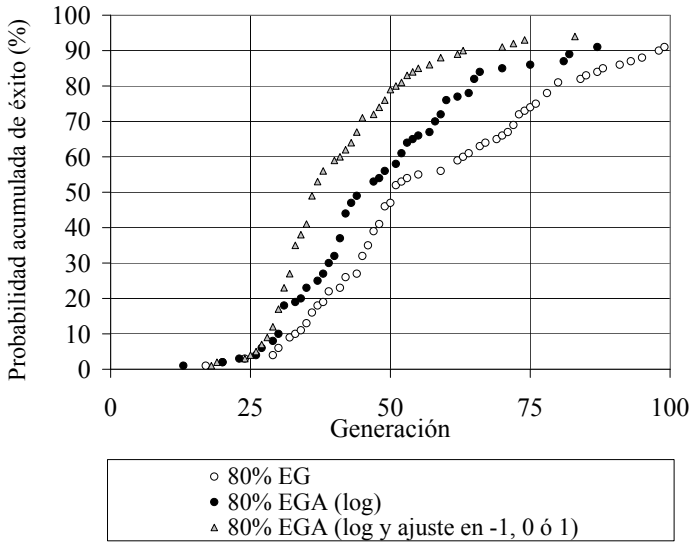

Figura 5.38: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 80 %, una tasa de recombinación del 90 % y sin uso de elitismo.

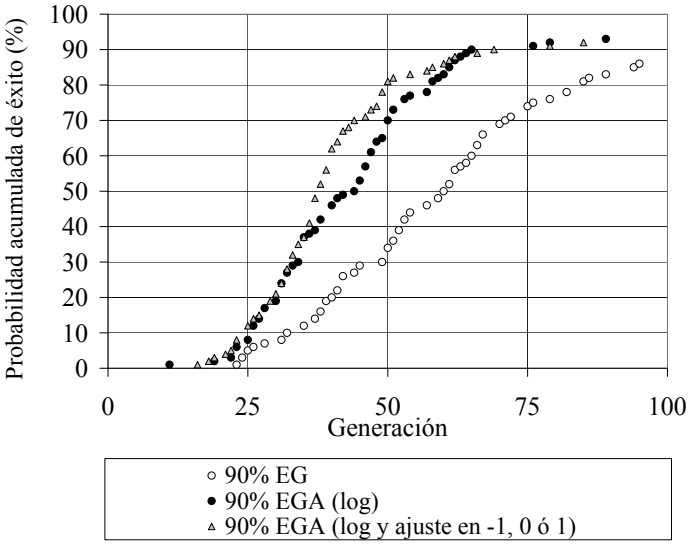

Figura 5.39: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 90 %, una tasa de recombinación del 90 % y sin uso de elitismo.

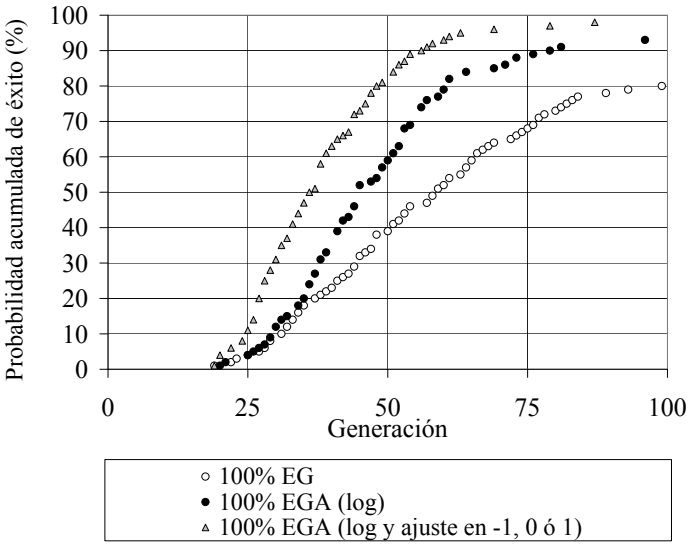

Figura 5.40: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de regresión simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 100 %, una tasa de recombinación del 90 % y sin uso de elitismo.

es capaz de encontrar ningún individuo correcto, imposibilitando el comienzo del proceso de evolución. Intuitivamente, este comportamiento es razonable. Imagínese el caso extremo en el que se obligara a los individuos a ajustar sin error todos los puntos para ser considerados semánticamente correctos. En este caso, el primer individuo generado ya sería una solución del problema y se habría sustituido la búsqueda genética por la inicialización aleatoria perdiendo las propiedades que garantiza el proceso evolutivo.

5.4. Segundo ejemplo: integración simbólica

El segundo ejemplo, de la misma manera que el primero, también está descrito en [O'Neill y Ryan (2003)] y por ello ha sido seleccionado para incluirlo en este trabajo. Es el problema de integración simbólica. En las siguientes secciones se describe el problema y se presentan las soluciones obtenidas con EG y EGA.

5.4.1. Planteamiento del problema

El problema de integración simbólica consiste en encontrar la expresión matemática de una función que es la integral de una determinada curva. En particular, la función cuya integral se busca es:

$$f(x) = \cos(x) + 2x + 1$$

y su integral es:

$$f(x) = \sin(x) + x^2 + x$$

La solución se aborda utilizando la misma estrategia que se describe en [O'Neill y Ryan (2003) y Koza (1992)], que consiste en reducir el problema a otro de regresión simbólica cuyo objetivo es encontrar la expresión de la función integrada, utilizando un conjunto de 21 puntos de ajuste en el intervalo $[0, 2\pi]$

5.4.2. Solución del problema con EG

La gramática independiente del contexto que se utiliza en este problema es ligeramente distinta a la utilizada en el primer ejemplo ya que incorpora el operador de división y la constante entera 1,0. Se define como $G = \{\Sigma_T, \Sigma_N, S, P\}$, donde el conjunto de terminales Σ_T es:

$$\Sigma_T = \{\sin, \cos, \exp, \log, +, -, *, /, x, 1{,}0, (,)\}$$

el conjunto Σ_N de no terminales es el siguiente:

$$\Sigma_N = \{expr, pre\_op, var\}$$

y el axioma,

$$S = expr$$

Por último, el conjunto de producciones P contiene las siguientes reglas:

(0)	$< expr >$	$\rightarrow$	$<expr> + <expr>$
(1)		$\mid$	$<expr> - <expr>$
(2)		$\mid$	$<expr> * <expr>$
(3)		$\mid$	$<expr> / <expr>$
(4)		$\mid$	$(<expr>)$
(5)		$\mid$	$<pre\_op> (<expr>)$
(6)		$\mid$	$<var>$

(0)	$<pre\_op>$	$\rightarrow$	sin
(1)		$\mid$	cos
(2)		$\mid$	exp
(3)		$\mid$	log

(0)	$<var>$	$\rightarrow$	x
(1)	$<var>$	$\rightarrow$	1,0

En cuanto a la función de *fitness*, se define el *raw fitness* como la suma del error en valor absoluto en 21 puntos de ajuste equiespaciados en el intervalo $[0, \pi]$, que son los puntos $x_0 = 0$, $x_1 = \pi/10$, $x_2 = 2\pi/10$, ..., $x_{20} = 2\pi$. La definición de *fitness ajustado* descrita en el ejemplo de regresión simbólica se utiliza también en este problema con el objetivo de que mejor individuo tenga el mayor valor de *fitness*, y así, asegurar el correcto funcionamiento de la selección proporcional al *fitness* para elegir a los progenitores en cada paso evolutivo. El *fitness ajustado* definido de esta manera se encuentra entre 0 y 1, y es mayor cuanto mejor sea el individuo.

Si en la creación de un individuo se consumen todos los codones de su genotipo y su fenotipo contiene aún símbolos no terminales, se le asigna el

peor valor de *fitness*, en este caso 0, y la presión selectiva lo hará desaparecer de la población.

Una vez definida la gramática independiente del contexto y la función de *fitness*, se establecen los valores de los parámetros del algoritmo según se muestra en tabla 5.7.

Tamaño de la población	500
Número máximo de genraciones	100
Número mínimo de codones (inicial)	1
Número máximo de codones (inicial)	10
Valor mínimo de codón	0
Valor máximo de codón	256
Operador de *wrapping*	inhabilitado
Elitismo	NO
Tasa de recombinación	90 %
Tasa de mutación [2]	0 %, 10 %, 20 %, ..., 100 %

Tabla 5.7: Parámetros de ejecución del algoritmo EG para el problema de integración simbólica.

[2]Cada tasa de mutación se corresponde con una de las 11 pruebas.

En la figura 5.41 se muestran los resultados de las 11 series de 100 ejecuciones independientes cada una, realizadas con los valores descritos para los parámetros, y correspondiendo cada serie a una tasa de mutación distinta.

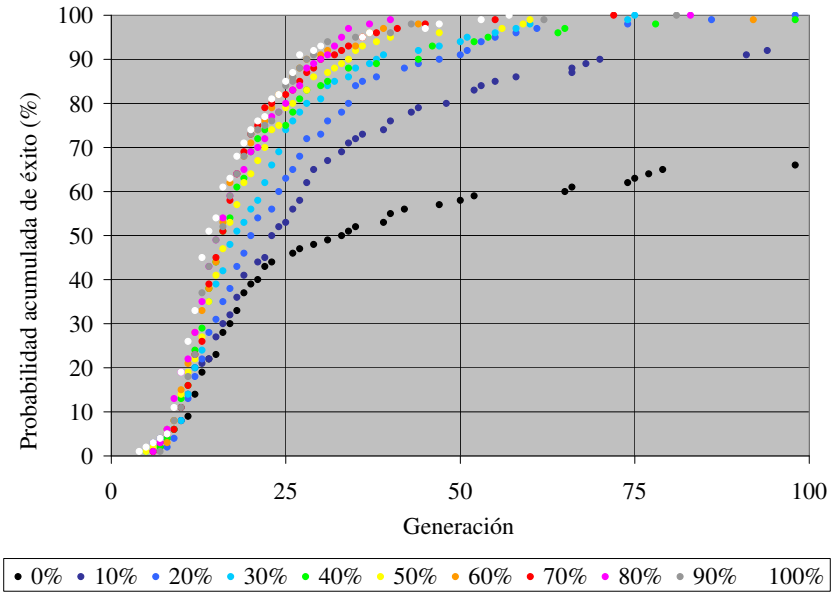

Figura 5.41: Probabilidad acumulada de éxito de EG para el problema de integración simbólica, calculada sobre 11 series de 100 ejecuciones independientes para 11 tasas diferentes de mutación, una tasa de recombinación del 90 % y sin uso de elitismo.

Se puede observar el incremento en el rendimiento a medida que aumenta la tasa de mutación. Comparando con las curvas de rendimiento obtenidas en el primer ejemplo, se observa que en este segundo problema, los valores de rendimiento son superiores de manera significativa mientras que la única diferencia entre ambos es una mínima modificación en la gramática y evidentemente, el cambio de la función objetivo de la regresión. Comparando las figuras 5.15 y 5.41 se puede observar que en el problema de regresión simbólica (fig. 5.15), la probabilidad de éxito acumulada en la generación 75 oscila aproximadamente en el intervalo $[35, 90]$, mientras que en el de integración simbólica (fig. 5.41), ya en la generación 25 se alcanza una horquilla similar.

5.4.3. Solución del problema con EGA

La transformación del problema de integración simbólica en uno de regresión simbólica, sugiere un diseño similar del sistema de atributos, que permita evaluar en los 21 puntos de evaluación las subexpresiones del fenotipo durante su construcción. Y con ese sistema de atributos, también parece lógico diseñar unas restricciones semánticas parecidas a las establecidas en el problema de regresión simbólica. Con estas pautas se han realizado dos experimentos con dos semánticas diferentes.

5.4.3.1. Primer experimento

En el primer experimento se diseña una gramática de atributos muy parecida a la utilizada en el primer experimento del problema de regresión simbólica. Tiene como objetivo rechazar aquellos individuos que contengan en su fenotipo subexpresiones que no estén definidas en alguno de los 21 puntos de evaluación de la función. En la gramática independiente del contexto que se ha utilizado para resolver el problema con EG, ésta condición de rechazo es cierta en dos circunstancias. En primer lugar cuando aparece el logaritmo de una expresión cuyo valor es menor o igual que 0, y en segundo lugar cuando el denominador de una división es 0. Como es habitual, si durante la construcción de un fenotipo, se detecta el incumplimiento de una de las dos restricciones semánticas, se interrumpe el proceso de generación del fenotipo, y el individuo es rechazado, es decir, no entra a formar parte de la población.

La gramática de atributos se define como $G_1 = \{\Sigma_T, \Sigma_N, S, P, K\}$, donde el conjunto de terminales Σ_T es:

$$\Sigma_T = \{\sin, \cos, \exp, \log, +, -, *, /, x, 1, 0, (,)\}$$

Ninguno de los símbolos terminales tiene atributos definidos y por este motivo, para simplificar la notación, se omiten los paréntesis que acompañan a cada símbolo en la definición formal de una gramática de atributos.

El conjunto de los símbolos no terminales Σ_N es define como:

$$\begin{aligned}\Sigma_N = \{ \quad & expr\ (\mathbb{R} \quad valor_i\ ,,\ i = 0, 1, \ldots, 20), \\ & var\ (\mathbb{R} \quad punto_i\ ,,\ i = 0, 1, \ldots, 20), \\ & pre\_op\ ([\ \text{“sin”},\ \text{“cos”},\ \text{“exp”},\ \text{“log”}] \quad funcion) \quad \}\end{aligned}$$

donde el atributo $valor_i$ almacena el valor de la subexpresión representada por el no terminal $<expr>$ en el punto de evaluación i, el atributo $punto_i$

guarda el valor del punto de evaluación i, y por último, el atributo *funcion* almacena una cadena de caracteres que define la función representada por el símbolo <*pre_op*>.

El axioma de la gramática de atributos es:

$$S = expr$$

No se define información global de la gramática de atributos, $K = \Phi$.

Por último, el conjunto P de producciones junto con sus acciones semánticas es el siguiente:

(0) <*expr*> → <*expr*>$_1$ + <*expr*>$_2$
{
DESDE $i = 0$ **HASTA** 20
<*expr*> .*valor*$_i$ =<*expr*>$_1$.*valor*$_i$+ <*expr*>$_2$.*valor*$_i$
}

(1) <*expr*> → <*expr*>$_1$ − <*expr*>$_2$
{
DESDE $i = 0$ **HASTA** 20
<*expr*> .*valor*$_i$ =<*expr*>$_1$.*valor*$_i$− <*expr*>$_2$.*valor*$_i$
}

(2) <*expr*> → <*expr*>$_1$ ∗ <*expr*>$_2$
{
DESDE $i = 0$ **HASTA** 20
<*expr*> .*valor*$_i$ =<*expr*>$_1$.*valor*$_i$∗ <*expr*>$_2$.*valor*$_i$
}

(3) <*expr*> → <*expr*>$_1$ / <*expr*>$_2$
{
DESDE $i = 0$ **HASTA** 20
SI (<*exp*>$_2$.*valor*$_i$ == 0)
ERROR SEMÁNTICO
SI NO <*expr*> .*valor*$_i$ =<*expr*>$_1$.*valor*$_i$/ <*expr*>$_2$.*valor*$_i$
}

(4) <*expr*> → (<*expr*>$_1$)
{
DESDE $i = 0$ **HASTA** 20
<*expr*> .*valor*$_i$ =<*expr*>$_1$.*valor*$_i$
}

(5) <*expr*> → <*pre_op*> (<*expr*>$_1$)
{
DESDE $i = 0$ **HASTA** 20
SI (<*pre_op*> .*funcion* ==“log”) **Y** (<*expr*>$_1$.*valor*$_i$ ≤ 0)
ERROR SEMÁNTICO
SI NO <*expr*> .*valor*$_i$ = **APLICAR** <*pre_op*> .*funcion*
A <*expr*>$_1$.*valor*$_i$
}

(6) <*expr*> → (<*var*>)
{
DESDE $i = 0$ **HASTA** 20
<*expr*> .$valor_i$ =<*var*> .$punto_i$
}

(0) <*pre_op*> → sin { <*pre_op*> .$funcion$ = "sin" }
(1) <*pre_op*> → cos { <*pre_op*> .$funcion$ = "cos" }
(2) <*pre_op*> → cos { <*pre_op*> .$funcion$ = "exp" }
(3) <*pre_op*> → cos { <*pre_op*> .$funcion$ = "log" }

(0) <*var*> → x {
<*var*> .$punto_0 = 0$
<*var*> .$punto_1 = \pi/10$
<*var*> .$punto_2 = 2\pi/10$
<*var*> .$punto_3 = 3\pi/10$
<*var*> .$punto_4 = 4\pi/10$
<*var*> .$punto_5 = 5\pi/10$
<*var*> .$punto_6 = 6\pi/10$
<*var*> .$punto_7 = 7\pi/10$
<*var*> .$punto_8 = 8\pi/10$
<*var*> .$punto_9 = 9\pi/10$
<*var*> .$punto_{10} = pi$
<*var*> .$punto_{11} = 11\pi/10$
<*var*> .$punto_{12} = 12\pi/10$
<*var*> .$punto_{13} = 13\pi/10$
<*var*> .$punto_{14} = 14\pi/10$
<*var*> .$punto_{15} = 15\pi/10$
<*var*> .$punto_{16} = 16\pi/10$
<*var*> .$punto_{17} = 17\pi/10$
<*var*> .$punto_{18} = 18\pi/10$
<*var*> .$punto_{19} = 19\pi/10$
<*var*> .$punto_{20} = 2\pi$
}
(1) <*var*> → 1,0
{
DESDE $i = 0$ **HASTA** 20
<*var*> .$punto_i$ = 1,0
}

El conjunto de pruebas realizado con la gramática G_1 tiene como objetivo poder comparar los resultados obtenidos, en términos de rendimiento, con los obtenidos para el mismo problema utilizando EG. Se han realizado 11 pruebas de 100 ejecuciones independientes cada una de ellas.

Los valores de los parámetros configurables del algoritmo son iguales a los utilizados en EG y se resumen en la siguiente tabla:

Tamaño de la población	500
Número máximo de genraciones	100
Número mínimo de codones (inicial)	1
Número máximo de codones (inicial)	10
Valor mínimo de codón	0
Valor máximo de codón	256
Operador de *wrapping*	inhabilitado
Elitismo	NO
Tasa de recombinación	90 %
Tasa de mutación [3]	0 %, 10 %, 20 %, ..., 100 %

Tabla 5.9: Parámetros de ejecución del algoritmo EGA para el problema de regresión simbólica.

En las figuras 5.42 - 5.52 se muestran los rendimientos obtenidos en las 11 pruebas, comparados con los rendimientos obtenidos para el mismo problema resuelto con EG (en estas figuras también se incluyen los rendimientos obtenidos en el experimento que se describe en la siguiente sección, que resuelve el mismo problema con una semántica más restrictiva). En las gráficas se puede observar que solamente en algunas de las 11 pruebas se supera el rendimiento obtenido con EG. Este resultado puede ser debido a que la semántica utilizada para descartar individuos también está, de alguna manera, implícita en la función de *fitness* de EG, ya que, si un individuo no se puede evaluar en algún punto, se penaliza con el peor *fitness*. Por lo tanto, aunque no es lo mismo descartar a un individuo e impedir que entre a formar parte de la población, que penalizarlo con la función *fitness*, podría ocurrir que el resultado en términos de rendimiento fuera similar, como de hecho parece que ha ocurrido en este ejemplo.

En la tabla 5.10 se muestran los cálculos relativos al número de individuos rechazados por incumplimiento de la restricción semántica establecida. De nuevo, se mantiene el orden de magnitud para la media y la desviación estándar. Se podría decir que, para el problema de integración simbólica,

[3] Cada tasa de mutación se corresponde con una de las 11 pruebas.

con la semántica definida en este primer experimento, en promedio, se rechazan 26 individuos en cada generación. Si se compara este valor con el obtenido en el primer experimento de regresión simbólica, 21 individuos, se observa que son similares, y parece razonable, ya que, las restricciones semánticas también lo son. Además, en el problema de integración simbólica, la semántica es ligeramente más restrictiva y el promedio de individuos rechazados es superior, como también parece lógico.

Tasa de mutación	media	desviación estándar
0 %	15	13
10 %	19	13
20 %	20	11
30 %	24	16
40 %	25	13
50 %	28	11
60 %	25	8
70 %	31	13
80 %	30	14
90 %	35	18
100 %	35	16

Tabla 5.10: Media y desviación estándar del número de individuos rechazados en cada generación del primer experimento de EGA del problema de integración simbólica.

5.4.3.2. Segundo experimento

El segundo experimento se realiza con una semántica más restrictiva de manera que a los individuos que ya se rechazaban en el primer experimento, se añaden también aquellos que no ajusten de manera exacta la función objetivo en al menos alguno los puntos 0, π ó 2π. La gramática de atributos es similar a la utilizada en el primer experimento, en concreto, el sistema de atributos es el mismo, pero modifican las acciones semánticas incorporando en las del axioma la nueva restricción. La comprobación del ajuste exacto en al menos alguno de los tres puntos mencionados sólo se puede realizar una vez terminada la generación del fenotipo, a diferencia del primer experimento en el que las restricciones semánticas son comprobables durante la construcción del mismo.

La gramática utilizada en este experimento se define como $G_2 =$

$\{\Sigma_T, \Sigma_N, S, P, K\}$, donde los conjuntos Σ_T, Σ_N, K y el axioma S coinciden con los de la gramática G_1, y el conjunto P de producciones se obtiene modificando el conjunto P de la gramática G_1 de manera que en todas las reglas del axioma se incorpora la comprobación del ajuste exacto en alguno de los puntos 0, π ó 2π. Como ejemplo, se muestra a continuación la acción semántica de la primera regla del axioma.

(0) <*expr*> → <*expr*>$_1$ + <*expr*>$_2$
{
DESDE $i = 0$ **HASTA** 20
<*expr*> .*valor*$_i$ =<*expr*>$_1$.*valor*$_i$+ <*expr*>$_2$.*valor*$_i$
SI (<*exp*> *ES LA RAÍZ DEL ÁRBOL DE DERIVACIÓN*) **Y**
(<*expr*> .*valor*$_0$! = 0) **Y**
(<*expr*> .*valor*$_{10}$! = $\pi^2 + \pi$) **Y**
(<*expr*> .*valor*$_{20}$! = $4\pi^2 + 2\pi$)
ERROR SEMÁNTICO
}

Se han realizado 11 pruebas de 100 ejecuciones independientes, similares a las realizadas en el primer experimento y sus resultados se muestran en las figuras 5.42 - 5.52. Aunque las pruebas se realizaron con un número máximo de generación igual a 100, en las gráficas sólo se muestran valores hasta la generación 50 porque en la mayoría de las 1100 ejecuciones se encontró una solución al problema antes de la generación 50. Se puede observar que en todos los casos el rendimiento obtenido en el segundo experimento, con la semántica más restrictiva, es superior al obtenido en el primero, con la semántica más relajada, y también claramente superior al obtenido con EG.

En cuanto al número de individuos rechazados, en la tabla 5.11 se muestran los resultados obtenidos. La media de individuos rechazados se mantiene estable para las distintas tasas de mutación, y el promedio es 128. Este es el valor más alto obtenido en todos los experimentos realizados, tanto de integración como de regresión simbólica. Este resultado es coherente ya que este último experimento es el que tiene la semántica más restrictiva.

Tasa de mutación	media	desviación estándar
0 %	34	20
10 %	104	38
20 %	113	40
30 %	125	37
40 %	127	33
50 %	135	31
60 %	141	27
70 %	152	34
80 %	154	24
90 %	164	29
100 %	164	24

Tabla 5.11: Media y desviación estándar del número de individuos rechazados en cada generación del segundo experimento de EGA del problema de integración simbólica.

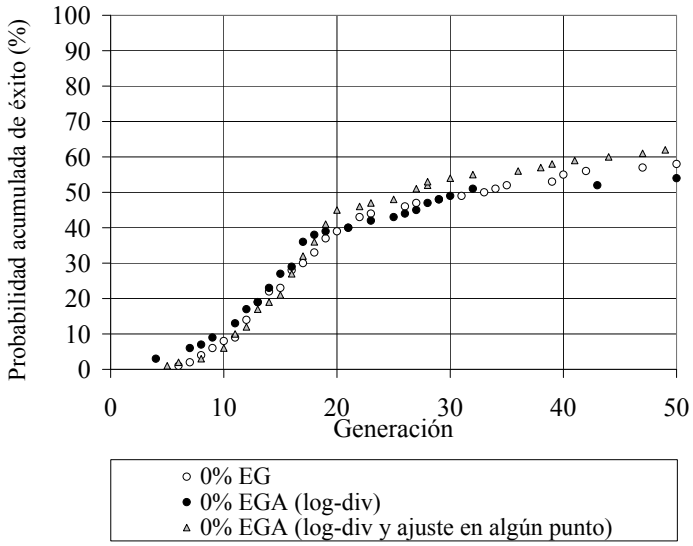

Figura 5.42: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de integración simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 0 %, una tasa de recombinación del 90 % y sin uso de elitismo.

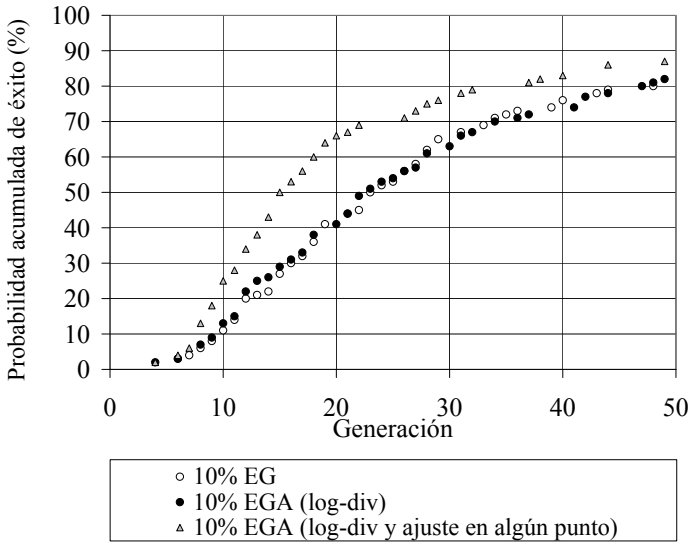

Figura 5.43: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de integración simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 10 %, una tasa de recombinación del 90 % y sin uso de elitismo.

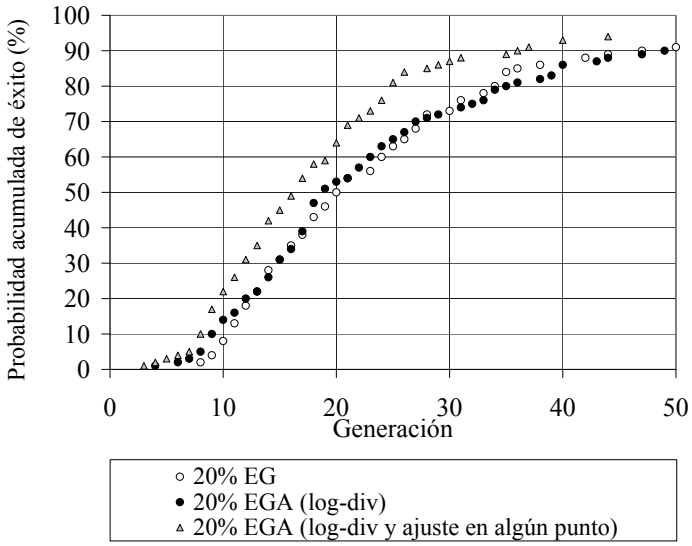

Figura 5.44: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de integración simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 20 %, una tasa de recombinación del 90 % y sin uso de elitismo.

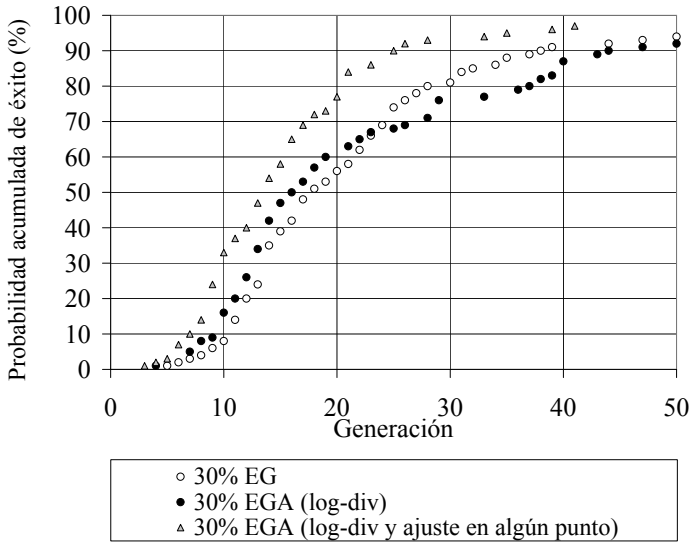

Figura 5.45: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de integración simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 30 %, una tasa de recombinación del 90 % y sin uso de elitismo.

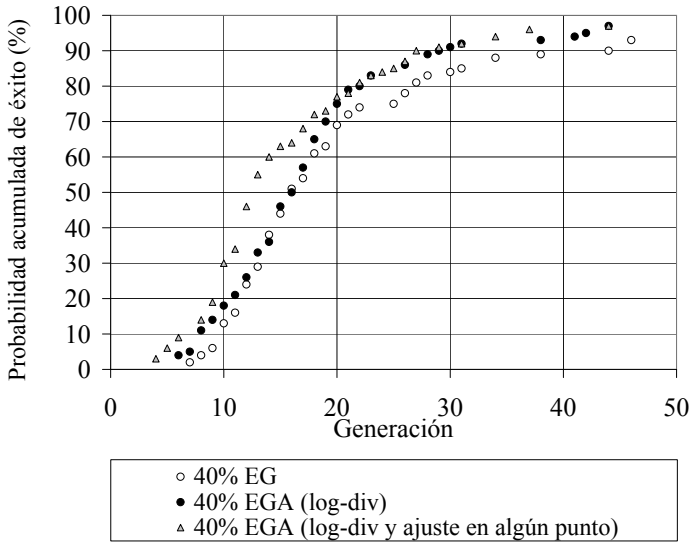

Figura 5.46: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de integración simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 40 %, una tasa de recombinación del 90 % y sin uso de elitismo.

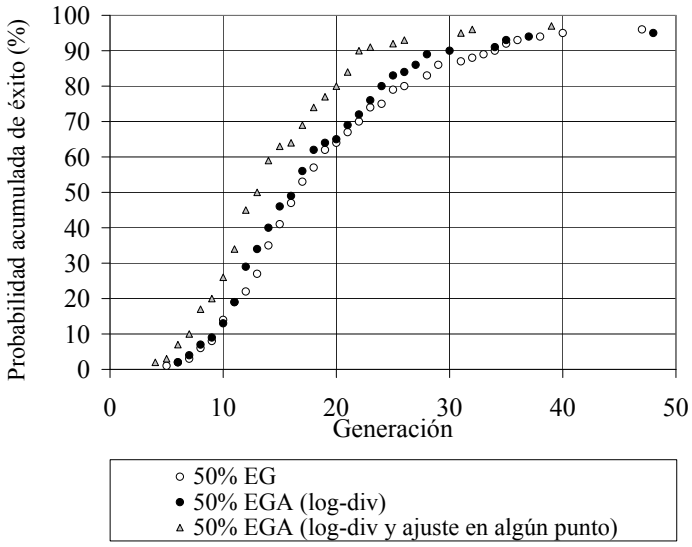

Figura 5.47: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de integración simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 50 %, una tasa de recombinación del 90 % y sin uso de elitismo.

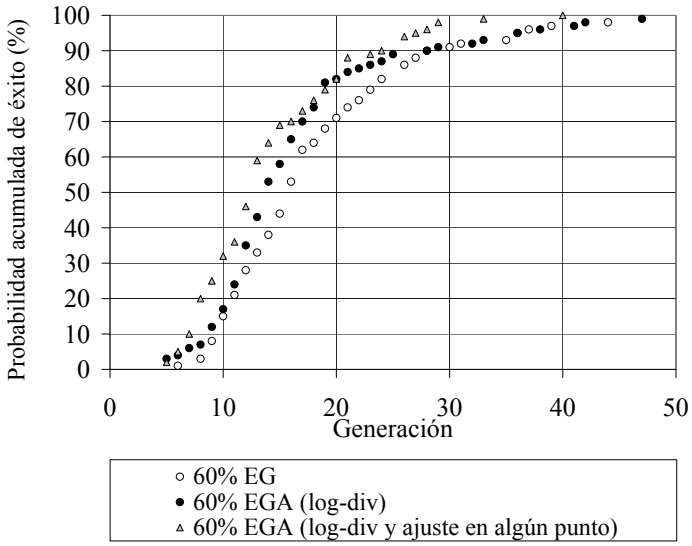

Figura 5.48: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de integración simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 60 %, una tasa de recombinación del 90 % y sin uso de elitismo.

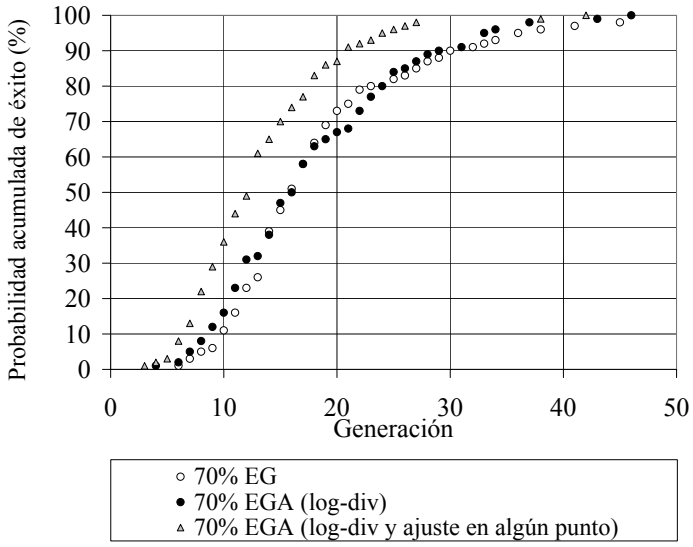

Figura 5.49: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de integración simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 70 %, una tasa de recombinación del 90 % y sin uso de elitismo.

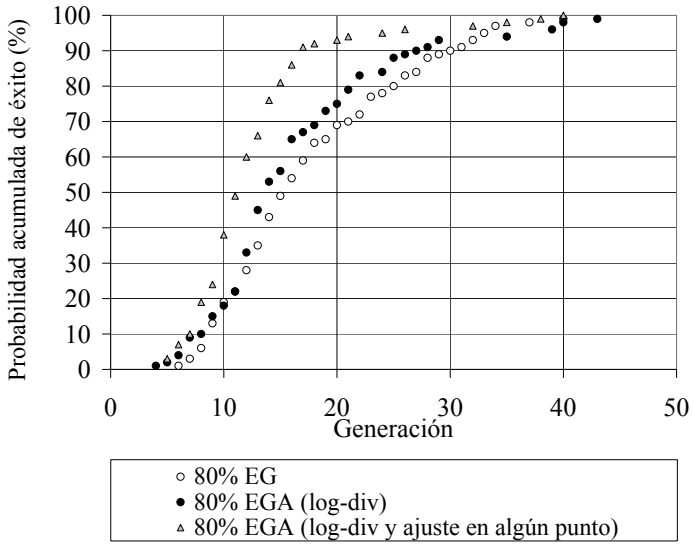

Figura 5.50: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de integración simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 80 %, una tasa de recombinación del 90 % y sin uso de elitismo.

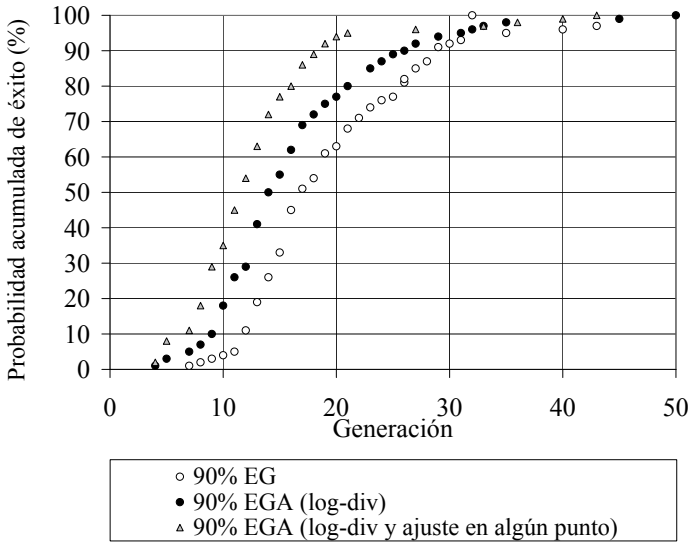

Figura 5.51: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de integración simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 90 %, una tasa de recombinación del 90 % y sin uso de elitismo.

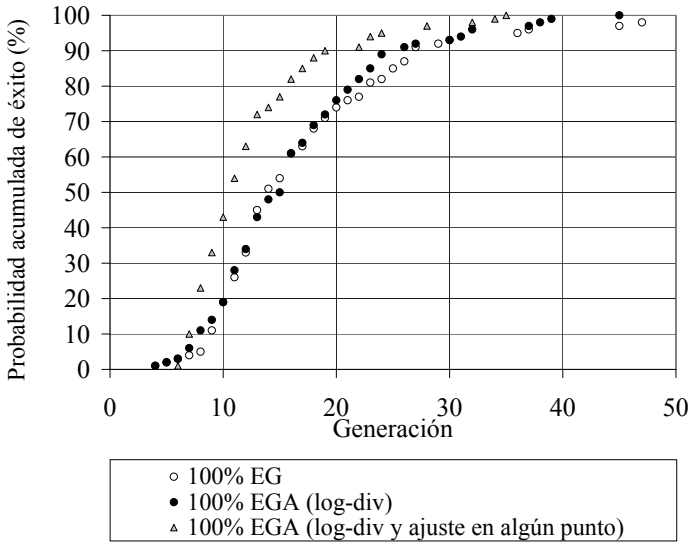

Figura 5.52: Comparativa de las probabilidades acumuladas de éxito de EG y EGA con dos niveles de semántica, para el problema de integración simbólica a partir de 100 ejecuciones independientes, una tasa de mutación del 100 %, una tasa de recombinación del 90 % y sin uso de elitismo.

Capítulo 6

Evolución con gramáticas de Christiansen

La segunda propuesta de esta tesis consiste en la incorporación de semántica a EG haciendo uso de gramáticas de Christiansen.

En este capítulo se detalla con profundidad la propuesta y se muestran los resultados de su aplicación en algunos problemas clásicos del área. De la misma manera que se hizo en el capítulo anterior, los problemas que se presentan se resuelven, por un lado con EG y por otro con la nueva propuesta de EG con semántica expresada con gramáticas de Christiansen para, así, poder realizar un estudio comparativo de los resultados y, si es posible, extraer conclusiones.

6.1. Objetivo

El objetivo de incorporar semántica a EG con gramáticas de Christiansen es el mismo que motivó el desarrollo del método EGA, proporcionar un mecanismo que permita durante el proceso evolutivo la generación de individuos sintáctica y semánticamente correctos. Sin embargo, aunque el objetivo es el mismo, este nuevo método permite diseñar soluciones sencillas, directas y "elegantes" para problemas cuya solución con EGA sería seguramente mucho más complicada e indirecta, y prácticamente inviable con EG.

6.2. El método

En este apartado se describe el método de Evolución con Gramáticas de Christiansen (a partir de ahora EGC) y se presenta un ejemplo de uso con

un problema diseñado específicamente para ello. Finalmente se plantea una solución, para el mismo problema del ejemplo, pero implementada con EGA.

6.2.1. Descripción

El método EGC consiste básicamente en sustituir la gramática de atributos de EGA por una gramática de Christiansen. Esta sustitución genera un cambio muy importante en el algoritmo de transformación de genotipo a fenotipo de EGA. Mientras que en EGA se utiliza la misma gramática de atributos durante toda la ejecución del algoritmo para derivar los no terminales del fenotipo en construcción, en EGC se utiliza la gramática de Christiansen almacenada en el primer atributo de cada no terminal. Como consecuencia de la variación o adaptabilidad intrínseca a las gramáticas de Christiansen, las reglas de derivación del mismo no terminal pueden ser distintas dependiendo del nodo del árbol en el que se encuentre dicho símbolo.

Otra consecuencia de la adaptabilidad de las gramáticas de Christiansen es que, durante la generación del fenotipo, puede ocurrir que un nodo pendiente de expandir herede, en su primer atributo, una gramática que no tenga reglas para el símbolo del propio nodo. Dado que la gramática almacenada en el primer atributo es precisamente la que se utiliza para derivar, el proceso de generación del fenotipo no podría continuar. Esta situación no es problemática ya que basta con añadir al algoritmo de transformación de genotipo a fenotipo una nueva regla (sería la regla 0) para comprobar, antes de hacer la expansión de un nodo, si existen producciones para su derivación. En el caso de no existir, se rechazaría el fenotipo en construcción de la misma manera que se hace en EGA.

Obviamente, el diseño de la gramática de Christiansen es un punto fundamental a la hora de plantear la solución de cualquier problema con EGC.

Por otra parte, la secuencia de construcción del árbol de derivación (en profundidad por la izquierda con retroseguimiento) durante el proceso de traducción de genotipo a fenotipo se mantiene igual para los dos métodos, así como los instantes en los que se evalúan los atributos de la gramática.

6.2.2. Ejemplo de uso

Para aclarar el funcionamiento del método se muestra un ejemplo de uso. Supóngase un problema en el que los fenotipos de los individuos de la población son sencillas expresiones aritméticas de variables del tipo:

$$a + b$$
$$a - b * (c + d)$$
$$x/y$$

Además, también pueden aparecer sumatorios con sus correspondientes variables con subíndices, como por ejemplo:

$$\sum_i a_i$$
$$\sum_j (x_j + y_j)$$
$$\sum_i \sum_k a_i * b_k$$

En un fenotipo pueden aparecer expresiones aritméticas sencillas combinadas con sumatorios, como por ejemplo:

$$x + y + \sum_i a_i$$
$$(b - c) * \sum_j (x_j + y_j)$$
$$\sum_i \sum_j (a_i + b/(x - y + z_j))$$

Por otra parte, se considera que un fenotipo es incorrecto si si se cumple alguna de las siguientes condiciones:

- En el cuerpo[1] de un sumatorio existe alguna variable con un subíndice que no ha sido definido como índice del sumatorio, o como índice de algún sumatorio externo, si lo hubiera. Por ejemplo, suponiendo que no están incluidos en ninguna expresión, los siguientes fenotipos serían incorrectos:

  $$\sum_i b_j$$
  $$\sum_j (w_j - z_k)$$
  $$\sum_i \sum_j (a_k * b_i * c_j)$$

- Aparece una variable con subíndice fuera de un sumatorio. Por ejemplo:

  $$a_i$$
  $$(w_j - z_k) * c$$

[1] Llamaremos cuerpo de un sumatorio a la expresión afectada por el mismo

Con el objetivo de simplificar la exposición del diseño de una gramática de Christiansen que permita la generación de fenotipos correctos, previamente se ha diseñado una gramática independiente del contexto, G_{I_SUM}, que genera expresiones del tipo descrito. Para ello, se ha adaptado la notación de manera que, el sumatorio:

$$\sum_i <expresion>$$

se representa con la siguiente cadena:

$$\sum[\ i\ ;\ <expresion>]$$

Además, las variables con subíndice del tipo a_i se representan por ai.

La gramática se define como $G_{I\_SUM} = \{\Sigma_N, \Sigma_T, P, S\}$, donde el conjunto de símbolos no terminales Σ_N es el siguiente:

$$\Sigma_N = \{expr, indice\_sum, indice\_var, var, letra\},$$

el conjunto Σ_T de terminales es el siguiente:

$$\Sigma_T = \{a, b, ..., z, \sum, (,), [,], ; \},$$

y el axioma

$$S = expr$$

Por último, el conjunto de reglas P contiene los elementos que se listan a continuación. Obsérvese que, para simplificar, los nombres de las variables se limitan a una letra y los de los subíndices permitidos al conjunto $\{i, j, k\}$.

(0) $<expr> \rightarrow <expr>_1 + <expr>_2$
(1) $<expr> \rightarrow <expr>_1 - <expr>_2$
(2) $<expr> \rightarrow <expr>_1 * <expr>_2$
(3) $<expr> \rightarrow <expr>_1 \ / <expr>_2$
(4) $<expr> \rightarrow (<expr>_1)$
(5) $<expr> \rightarrow \sum[<indice\_sum>; <expr>_1]$
(6) $<expr> \rightarrow <var>$

(0) $<var> \rightarrow <letra> <indice\_var>$

(0) $<indice\_sum> \rightarrow i$
(1) $\quad | \quad j$
(2) $\quad | \quad k$

$$
\begin{array}{llll}
(0) & <indice\_var> & \rightarrow & i \\
(1) & & | & j \\
(2) & & | & k \\
(3) & & | & \lambda
\end{array}
$$

$$
(0) \quad <letra> \quad \rightarrow \quad a|b|...|z
$$

La gramática G_{I_SUM} genera cadenas adecuadas a la definición del fenotipo del problema que nos ocupa, excepto en un aspecto, que es precisamente dependiente del contexto. Este aspecto se refiere a la restricción que obliga a que, en cualquier fenotipo correcto, no pueda aparecer ninguna variable con un subíndice que no haya sido previamente definido. Efectivamente, con la gramática G_{I_SUM} se pueden generar fenotipos incorrectos como los ejemplos mostrados anteriormente.

Es sencillo diseñar una gramática de Christiansen que permita generar fenotipos correctos. En este caso se hará a partir de la gramática independiente del contexto G_{I_SUM}. Dado que las gramáticas de Christiansen son una extensión de las gramáticas de atributos, hay que definir los atributos de cada símbolo, las producciones junto con sus acciones semánticas conducentes a la evaluación de los atributos, y la información global. Evidentemente, hay que tener en cuenta que el primer atributo de un no terminal es la gramática de Christiansen que contiene las reglas para derivar dicho símbolo.

Antes de proponer la definición formal de la gramática de Christiansen, plantearemos de manera informal la idea principal en la que se fundamenta su diseño. En la gramática G_{I_SUM} se puede observar que las variables con índice se generan con la producción:

$$
(0) \quad <var> \quad \rightarrow \quad <letra> <indice\_var>
$$

donde el índice de la variable se obtiene de la derivación del no terminal $<indice\_var>$. Por lo tanto, para conseguir el objetivo final (que sólo se puedan derivar variables con índices previamente definidos) bastaría con que la gramática almacenada en el primer atributo del símbolo $<indice\_var>$ contuviera solamente las producciones correspondientes a los índices que hasta ese momento hayan sido declarados. Esto se puede conseguir, por ejemplo, si:

1. En el inicio de la construcción del fenotipo, sólo existe la producción lambda para el no terminal $<indice\_var>$. Es decir, la gramática almacenada en el primer atributo del axioma sólo contiene la producción vacía del símbolo $<indice\_var>$.

2. Cada vez que se deriva un índice con una de las producciones del no terminal <*indice_sum*>, se añade a la gramática la producción correspondiente del no terminal <*indice_var*>. Por ejemplo si se deriva un índice utilizando la producción <*indice_sum*>→ i entonces se añade la producción <*indice_var*>→ i para poder derivar desde ese momento variables con subíndice i. La pregunta es, ¿a qué gramática se añade la nueva producción del símbolo <*indice_var*>? Solamente puede haber una respuesta: obviamente a la gramática asignada al primer atributo del no terminal <*indice_var*> correspondiente.

3. Se establece un mecanismo para ello, es decir, para que el no terminal <*indice_var*> disponga en su primer atributo de la gramática que ha sido modificada según se describe en el punto anterior, y que por lo tanto sólo se puedan derivar variables con índices previamente definidos. Este mecanismo debe ser implementado mediante la herencia y síntesis de atributos en el proceso de diseño de la gramática. En particular, en nuestro ejemplo:

   a) El no terminal <*indice_sum*> tiene un segundo atributo de tipo gramática de Christiansen, que es sintentizado y cuyo valor resulta de las acciones descritas en el punto 2.

   b) En la producción (5) del no terminal <*expr*>, que corresponde a la expresión sumatorio, el símbolo <*expr*>$_1$ hereda en su primer atributo, la gramática que ha sintetizado su hermano <*indice_sum*>.

   c) En la producción (6) del no terminal $<expr>$, en cuya parte derecha aparece solamente el símbolo <*var*>, éste hereda en su primer atributo, la gramática que su padre heredó en su momento, y que almacena en su primer atributo.

   d) En la única producción del símbolo $<var>$, el no terminal <*indice_var*> de su parte derecha, hereda en su primer atributo la gramática almacenada en el primer atributo de<*var*>.

En la figura 6.1 se muestra la inicialización, modificación y propagación de las reglas del no terminal <*indice_var*> en el árbol de derivación del fenotipo $\sum_i a_i$. Se puede observar que en la parte superior izquierda de los nodos de los no terminales aparece un recuadro. Contiene las reglas del símbolo <*indice_var*> que están presentes en la gramática de Christiansen que es el primer atributo del símbolo. Se puede observar en la figura que el rectángulo izquierdo de la raíz sólo contiene la producción λ de <*indice_var*>. El

nodo del símbolo $<indice\_sum>$ tiene otro recuadro en su parte superior derecha. Contiene las reglas del símbolo $<indice\_var>$ que están disponibles en la gramática de Christiansen almacenada en su segundo atributo. Esta segunda gramática es un atributo sintetizado, cuyo valor, según se explicó anteriormente, se obtiene añadiendo a la gramática del primer atributo la regla $<indice\_var> \rightarrow i$. El nodo del símbolo $<letra>$ no tiene recuadro como los demás porque para su derivación es irrelevante qué reglas de $<indice\_var>$ están disponibles en su primer atributo, y tampoco tiene hijos no terminales a los que deba transmitir esa información, como es el caso de los nodos correspondientes a $<expr>$ y $<var>$. Los números que acompañan a los recuadros indican el orden en el que se producen las actualizaciones de las gramáticas a las que corresponden las producciones que aparecen en los recuadros.

Una vez expuesta la idea básica de la gramática de Christiansen del ejemplo que nos ocupa, su definición formal sería la siguiente.

Se define la gramática de Christiansen $G_{C\_SUM} = \{\Sigma_T, \Sigma_N, S, P, K\}$, donde el conjunto de terminales Σ_T es:

$$\Sigma_T = \{a, b, ..., z, \sum, (,), [,], ;\}$$

Ninguno de los símbolos terminales tiene atributos definidos y por este motivo, para simplificar la notación, se omiten los paréntesis que acompañan a cada símbolo en la definición formal de una gramática.

El conjunto de los símbolos no terminales Σ_N es define como:

$$\begin{aligned} \Sigma_N = \{ \quad & expr\ (GC\ \ g_h), \\ & indice\_sum\ (GC\ \ g_h, GC\ \ g_s), \\ & indice\_var\ (GC\ \ g_h), \\ & var\ (GC\ \ g_h), \\ & letra\ (GC\ \ g_h)\} \end{aligned}$$

Las siglas GC significan Gramática de Christiansen y se utilizan para indicar el tipo de los atributos. Como puede observarse, sólo $<indice\_sum>$ tiene un segundo atributo, también de tipo GC. Esto es así porque su expansión desencadena la incorporación de la correspondiente regla de $<indice\_var>$.

El axioma de la gramática es:

$$S = expr$$

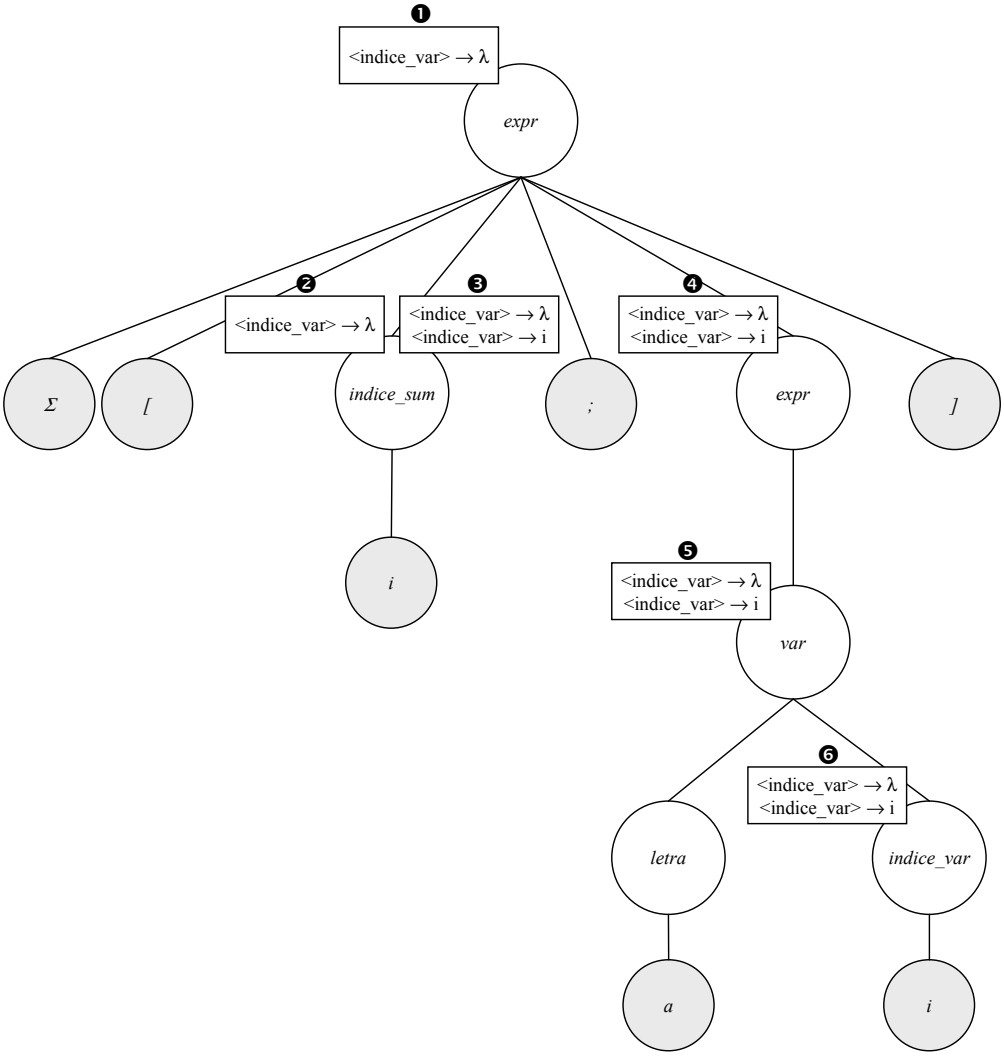

Figura 6.1: Inicialización, modificación y propagación de las reglas del no terminal <*indice_var*>.

El conjunto P de producciones está formado por las siguientes reglas con sus acciones semánticas:

(0) $<expr> \rightarrow <expr>_1 + <expr>_2$
{
$<expr>_1 .g_h = <expr> .g_h$
$<expr>_2 .g_h = <expr> .g_h$
}

(1) $<expr> \rightarrow <expr>_1 - <expr>_2$
{
$<expr>_1 .g_h = <expr> .g_h$
$<expr>_2 .g_h = <expr> .g_h$
}

(2) $<expr> \rightarrow <expr>_1 * <expr>_2$
{
$<expr>_1 .g_h = <expr> .g_h$
$<expr>_2 .g_h = <expr> .g_h$
}

(3) $<expr> \rightarrow <expr>_1 / <expr>_2$
{
$<expr>_1 .g_h = <expr> .g_h$
$<expr>_2 .g_h = <expr> .g_h$
}

(4) $<expr> \rightarrow (<expr>_1)$
{
$<expr>_1 .g_h = <expr> .g_h$
}

(5) $<expr> \rightarrow \sum[<indice\_sum>; <expr>_1]$
{
$<indice\_sum> .g_h = <expr> .g_h$
$<expr>_1 .g_h = <indice\_sum> .g_s$
}

(6) $<expr> \rightarrow <var>$
{
$<var> .g_h = <expr> .g_h$
}

(0) $<var> \rightarrow <letra> <indice\_var>$
{
$<letra> .g_h = <var> .g_h$
$<indice\_var> .g_h = <var> .g_h$
}

(0) $<indice\_sum> \rightarrow i$
{
$<indice\_sum>.g_s = <indice\_sum>.g_h \cup \{<indice\_var> \rightarrow i\}$
}
(1) $<indice\_sum> \rightarrow j$
{
$<indice\_sum>.g_s = <indice\_sum>.g_h \cup \{<indice\_var> \rightarrow j\}$
}
(2) $<indice\_sum> \rightarrow k$
{
$<indice\_sum>.g_s = <indice\_sum>.g_h \cup \{<indice\_var> \rightarrow k\}$
}

(0) $<indice\_var> \rightarrow \lambda \quad \{\}$

(0) $<letra> \rightarrow a \quad \{\}$
(1) $<letra> \rightarrow b \quad \{\}$
(2) $<letra> \rightarrow c \quad \{\}$
... ...

Por último, la gramática no tiene información global, es decir, $K = \Phi$.

A continuación se detalla la evolución del algoritmo de transformación de genotipo a fenotipo utilizando la gramática G_{C_SUM} a partir del siguiente genotipo:

40	3	75	100	23	6	0	7	41	1	8	47	42	6

El algoritmo de transformación de genotipo a fenotipo de EGC comienza con la construcción del nodo del árbol de derivación del fenotipo con el axioma de la gramática. Se asigna al primer atributo del axioma la gramática de Christiansen diseñada para el problema. En la figura 6.2 se muestra el árbol de derivación después del primer paso del algoritmo.

Puede observarse que se mantiene la representación que se utilizó para los árboles de derivación en el algoritmo de transformación de genotipo a fenotipo de EGA. Cada nodo tiene asignado un recuadro que contiene los valores de sus atributos. Para cada atributo se especifica su nombre y su valor. Esta representación se complica en EGC debido a que los atributos pueden ser gramáticas de Christiansen (el primer atributo de manera obligatoria) y sería demasiado pesado escribir las gramáticas completas. Por ello, en cada ejemplo, se intentará simplificar la información que aparece en

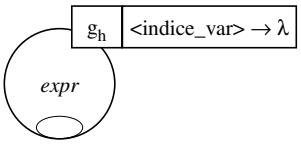

Figura 6.2: Árbol de derivación de EGC a partir de la gramática G_{C_SUM}, después de la construcción del nodo raíz.

el árbol de cada atributo. En particular, en el ejemplo que nos ocupa, la gramática de Christiansen solamente adapta las producciones del símbolo <*indice_var*> así que, en lugar de escribir la gramática completa únicamente se escriben las reglas aplicables al símbolo <*indice_var*>. También, en cada nodo correspondiente a un no terminal que ya ha sido expandido, se dibuja en una elipse interna el valor del codón utilizado para su expansión. El último nodo expandido se puede distinguir porque tiene una estrella en su interior. Por último, los nodos hoja correspondientes a los terminales de la cadena derivada se dibujan sombreados.

El algoritmo continúa con la expansión de la raíz y el primer codón del genotipo, 40. Como en la gramática que se utiliza para la derivación, la almacenada en el primer atributo del símbolo, el axioma tiene 7 reglas y $40 \ mod \ 7 = 5$, se utiliza la producción (5) para realizar la expansión aplicando la regla 1a) del algoritmo y obtener el árbol de derivación que se muestra en la figura 6.3.

En la acción semántica de la regla (5) del axioma se especifica que el símbolo <*indice_sum*> hereda en su primer atributo la gramática del primer atributo de su nodo padre. Por lo tanto, se aplica la regla 1b) del algoritmo, los atributos heredados de padres a hijos se calculan en el momento de la expansión del padre. El resultado de aplicar la regla 1b) se puede ver en la figura 6.3, en la que, como es habitual, se representa la herencia de atributos con una flecha punteada.

El proceso continúa con la expansión del nodo más profundo y situado más a la izquierda, que es el correspondiente al símbolo <*indice_sum*>. El codón actual vale 3, y la gramática del primer atributo de <*indice_sum*> contiene 3 producciones para su derivación. Se aplica la regla (0) ya que

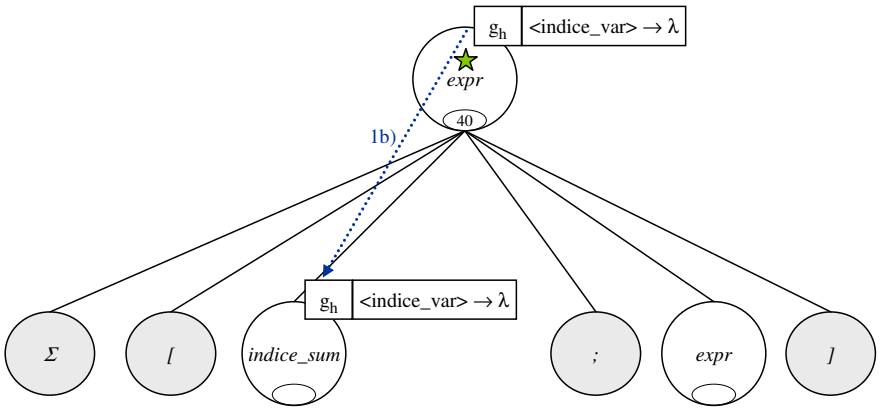

Figura 6.3: Árbol de derivación de EGC a partir de la gramática G_{C_SUM}, después de la expansión del nodo raíz.

$3 \; mod \; 3 = 0$. El resultado de la expansión se muestra en la figura 6.4. El nodo que se acaba de expandir, sólo tiene un hijo, que además es un terminal, por lo tanto, se aplica la regla 2b) relativa al cálculo de los atributos sintetizados, si los hubiere. En este caso, sí hay atributos sintetizados según se indica en la acción semántica de la regla aplicada:

$$
\begin{array}{lll}
(0) & <indice\_sum> & \rightarrow \quad i \\
 & \{ & \\
 & \quad <indice\_sum>.g_s = <indice\_sum>.g_h \cup \{<indice\_var> \rightarrow i\} & \\
 & \} &
\end{array}
$$

En la figura 6.4 se representa con una flecha de trazo continuo el cálculo del atributo sintetizado g_s del símbolo $<indice\_sum>$. Este atributo se calcula añadiendo una regla nueva para el símbolo $<indice\_var>$.

El nodo del símbolo $<indice\_sum>$ es el tercer hijo de la raíz del árbol, y tiene hermanos a su derecha. Su terminación desencadena la aplicación de la regla 2a) relativa a la evaluación de atributos heredados entre hermanos. En concreto, el primer atributo del quinto hijo del nodo raíz, que corresponde al no terminal $<expr>$, se calcula a partir del segundo atributo de su hermano $<indice\_sum>$. La evaluación del atributo heredado del símbolo $<expr>$, se realiza según se especifica en la acción semántica de la regla que se ha utilizado para la expansión del axioma, que en este caso es la regla (5) y tiene el siguiente aspecto:

(5) $<expr> \rightarrow \sum[<indice\_sum>; <expr>_1]$
{
$<indice\_sum>.g_h = <expr>.g_h$
$<expr>_1.g_h = <indice\_sum>.g_s$
}

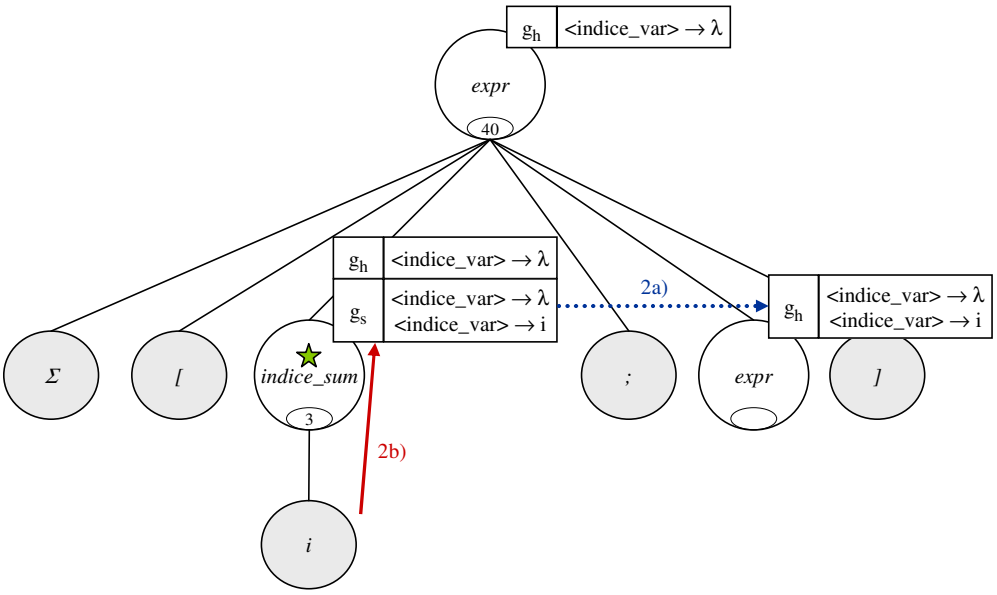

Figura 6.4: Árbol de derivación de EGC a partir de la gramática G_{C_SUM}, después de la expansión de la primera aparición del símbolo $<indice\_sum>$.

El siguiente nodo que se expande es el correspondiente a la segunda aparición del no terminal <*expr*> y el codón actual vale 75. Dado que el símbolo <*expr*> tiene siete producciones, se utilizará la producción (5) ya que 75 *mod* 7 = 5. Aplicando la regla 1a) del algoritmo se añaden al árbol los nodos hijos. También se aplica la regla 1b) dado que existe herencia de atributos, en concreto, el tercer hijo, que corresponde al no terminal <*indice_sum*>, hereda en su primer atributo la gramática que su padre heredó también en su primer atributo. En la figura 6.5 se muestra el estado del árbol de derivación después de esta expansión.

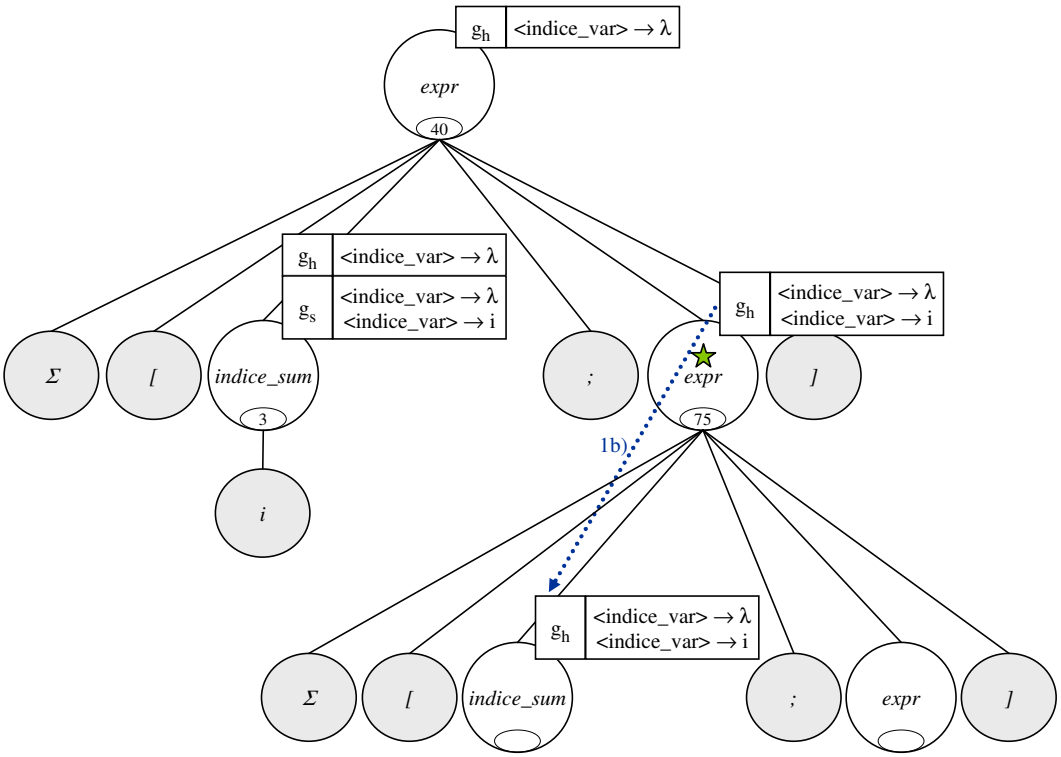

Figura 6.5: Árbol de derivación de EGC a partir de la gramática G_{C_SUM}, después de la expansión de la segunda aparición del símbolo <*expr*>.

El proceso continúa con el nodo correspondiente a la segunda aparición del símbolo <*indice_sum*>, y el codón actual, 100. Como la gramática del primer atributo de <*indice_sum*> contiene 3 producciones para su derivación. Se aplica la regla (1) ya que 100 *mod* 3 = 1. El resultado de la expansión se

muestra en la figura 6.6. La síntesis del segundo atributo de <*indice_sum*>, y su propagación por herencia a su hermano, que es el nodo de la tercera aparición de <*expr*>, ya se detalló en la descripción de la expansión de la primera aparición de <*indice_sum*>.

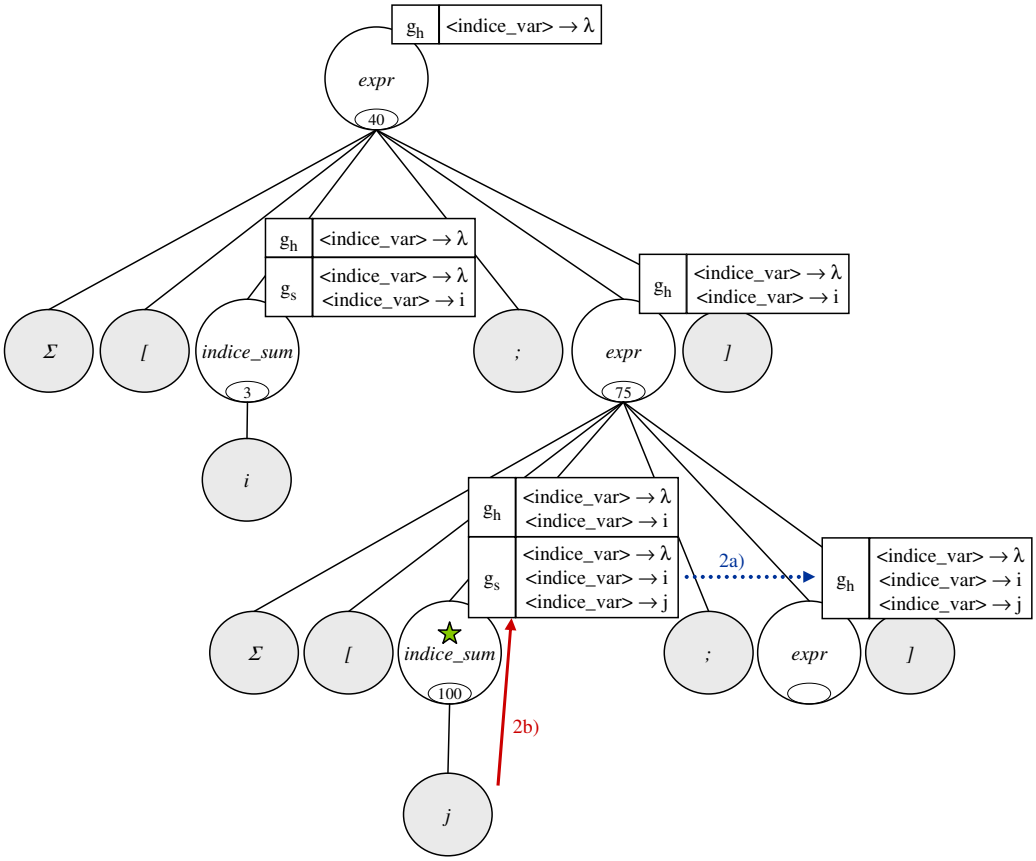

Figura 6.6: Árbol de derivación de EGC a partir de la gramática G_{C_SUM}, después de la expansión de la segunda aparición del símbolo <*indice_sum*>.

El nodo más profundo y más a la izquierda pendiente de expandir es el de la tercera aparición del símbolo <*expr*>. El codón actual es 23, y dado que la gramática del primer atributo del símbolo tiene 7 reglas, el nodo se expande con la regla (2) porque $23 \; mod \; 7 = 2$. La acción semántica de la regla usada en la expansión especifica que los dos símbolos <*expr*> de la parte derecha heredan en su primer atributo la gramática del primer atributo de su

nodo padre. Por lo tanto, se aplica la regla 1b) del algoritmo (los atributos heredados de padres a hijos se calculan en el momento de la expansión del padre). El resultado de aplicar la regla 1b) se puede ver en la figura 6.7, en la que, como es habitual, se representa la herencia de atributos con una flecha punteada. En la misma figura se puede observar que el último nodo expandido está conectado a un triángulo. Representa al subárbol construido antes de la expansión del nodo, y su contenido completo aparece en la figura 6.6. El objetivo de sustituir subárboles por triángulos es la simplificación de las figuras.

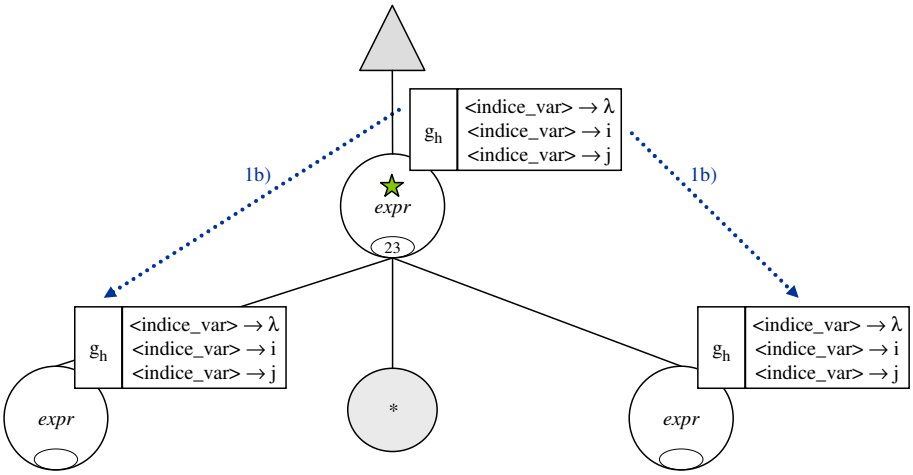

Figura 6.7: Subárbol de derivación de EGC a partir de la gramática G_{C_SUM}, después de la expansión de la tercera aparición del símbolo *<expr>*.

El siguiente nodo que se expande corresponde también al símbolo *<expr>*, y el codón actual es 6. La producción que se utiliza para la expansión es la (6) ya que 6 *mod* 7 = 6 y aplicando la regla 1a) se obtiene el árbol de la figura 6.8. En cuanto a evaluación de atributos, el único símbolo de la parte derecha, *<var>*, hereda en su primer atributo la gramática almacenada en el primer atributo del símbolo de la parte izquierda de la regla. Esta herencia queda reflejada con una flecha punteada en la figura 6.8.

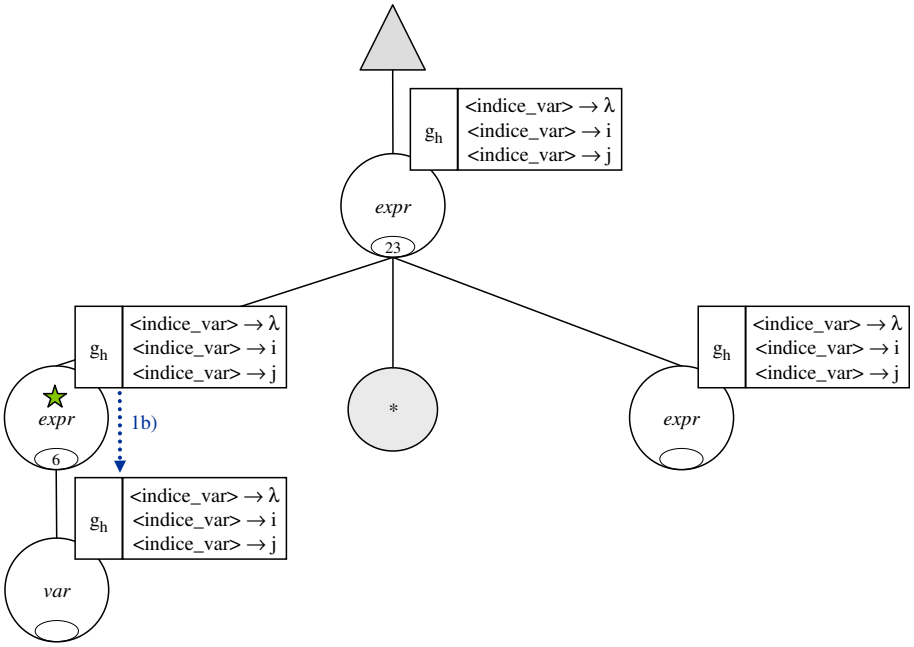

Figura 6.8: Subárbol de derivación de EGC a partir de la gramática G_{C_SUM}, después de la expansión de la cuarta aparición del símbolo *<expr>*.

El siguiente nodo que el algoritmo selecciona para expandir es el correspondiente al no terminal *<var>*. Éste símbolo solamente tiene una producción y por lo tanto no se consume codón. Aplicando la regla 1a) del algoritmo se añaden al árbol los dos nodos hijos. También se aplica la regla 1b) dado que existe herencia de atributos, en concreto, los dos hijos heredan en su primer atributo la gramática del primer atributo de su padre. En la figura 6.9 se muestra el estado del árbol de derivación después de la última operación de expansión. El óvalo situado dentro de los nodos que almacena el codón utilizado en la expansión, en el caso de no consumirse codón, aparece en negro.

El algoritmo continúa con la expansión del nodo correspondiente al símbolo *<letra>* y el codón 0. La aplicación de la regla 1a) con la producción (0) del no terminal *<letra>* da lugar al árbol de derivación que se muestra en la figura 6.10. La regla utilizada en la expansión contiene en su parte derecha un único símbolo, el terminal a. El procesamiento del nodo de a termina con su propia creación por el hecho de ser un nodo hoja, y al ser el único hijo, también termina el procesamiento de su nodo padre, el del símbolo *<letra>*.

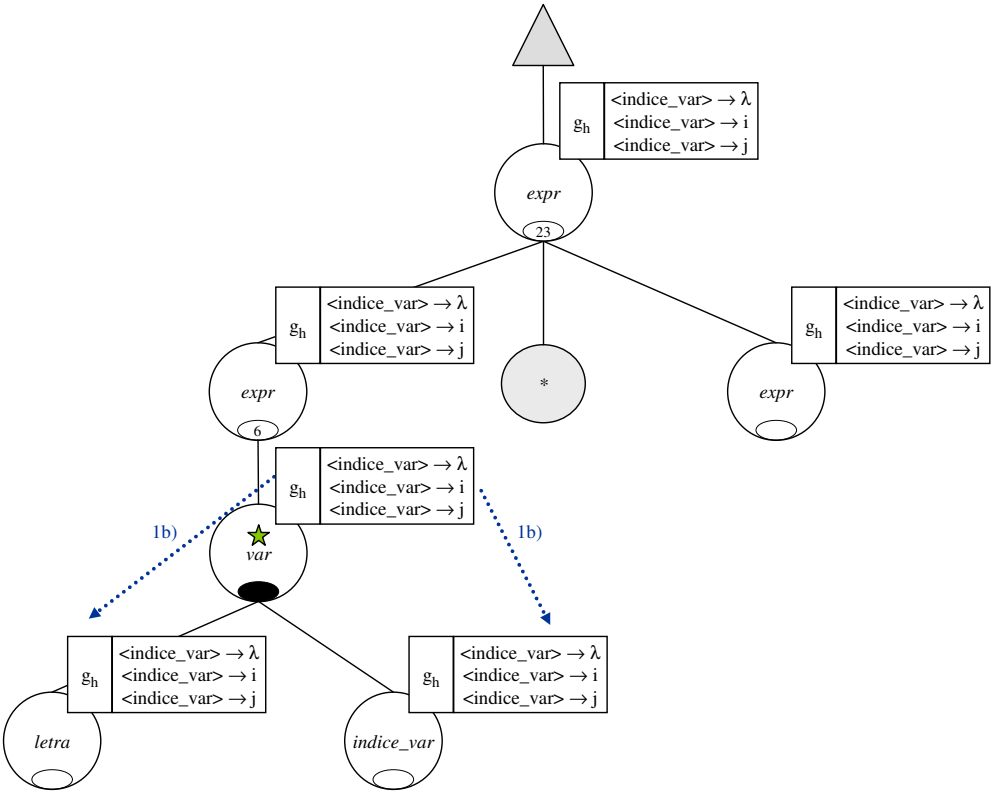

Figura 6.9: Subárbol de derivación de EGC a partir de la gramática G_{C_SUM}, después de la expansión de la primera aparición del símbolo *<var>*.

En este momento se evaluarían atributos heredados entre hermanos, si los hubiera, pero en este caso no procede porque en la acción semántica de la regla con la que se expandió el nodo de <*var*> sólo se especifica herencia de atributos.

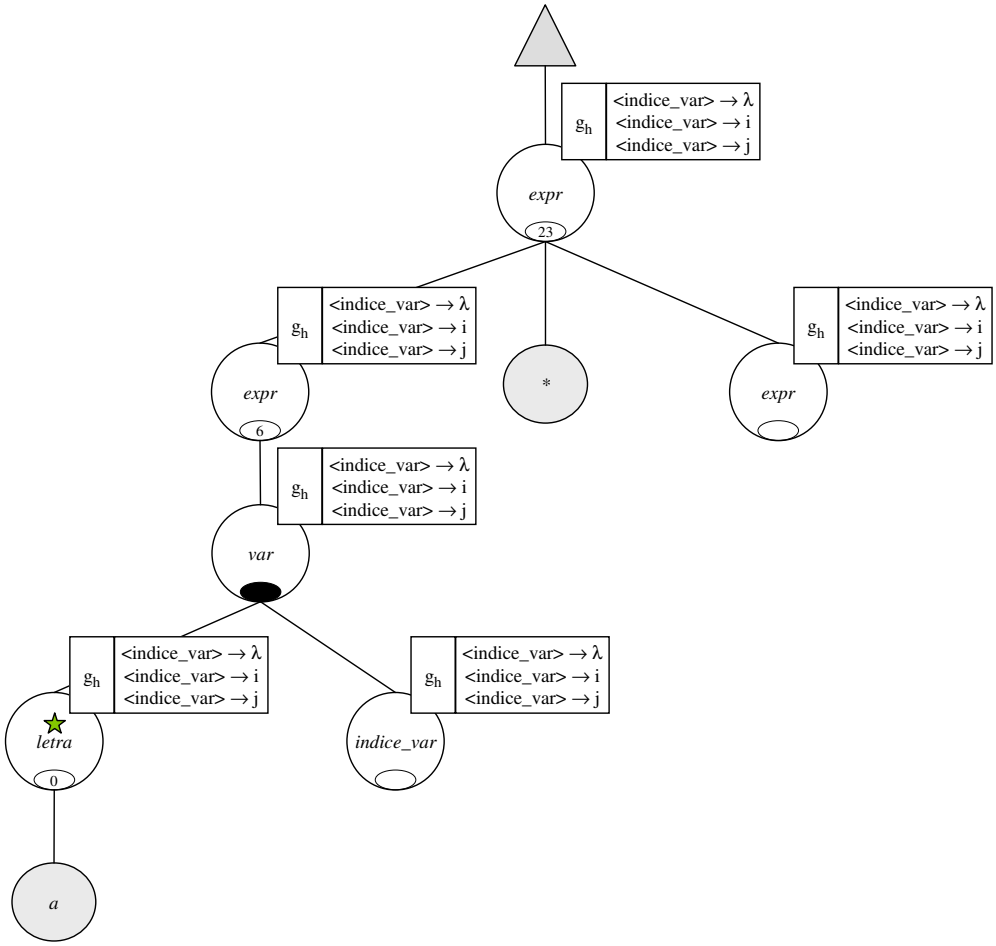

Figura 6.10: Subárbol de derivación de EGC a partir de la gramática G_{C_SUM}, después de la expansión de la primera aparición del símbolo <*letra*>.

El siguiente nodo que se expande es el del símbolo <*indice_var*>. En la gramática de Christiansen diseñada para el ejemplo, este símbolo es el único cuyas producciones pueden variar durante el proceso de construcción del fenotipo. Efectivamente, las derivaciones del símbolo <*indice_sum*> produ-

cen la incorporación de nuevas reglas para <*indice_var*>. En la figura 6.10, se puede ver en el recuadro asociado al nodo de <*indice_var*> las reglas disponibles para su expansión.

El codón actual es 7, por lo tanto, se utiliza para la expansión la regla (1) ya que 7 *mod* 3 = 1. En este paso del algoritmo no hay ni herencia ni síntesis de atributos. En la figura 6.11 se muestra el resultado de la última expansión.

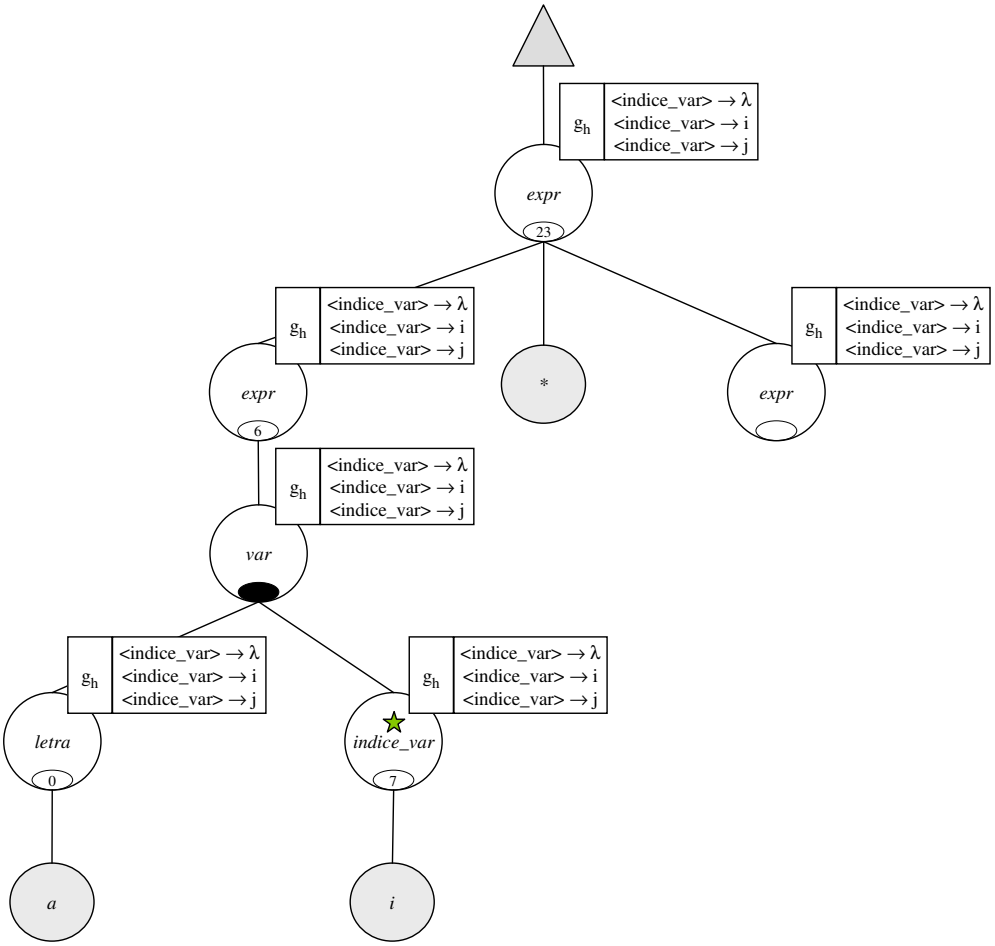

Figura 6.11: Subárbol de derivación de EGC a partir de la gramática G_{C_SUM}, después de la expansión de la primera aparición del símbolo <*indice_var*>.

El algoritmo continúa por el nodo del no terminal <*expr*> situado a la

derecha del símbolo $*$. Se omite la explicación de la evolución del algoritmo a partir de este nodo porque los siguientes codones (41, 1 y 8) generan un subárbol similar al ya generado a partir del símbolo $<expr>$ situado a la izquierda del símbolo $*$. La única diferencia entre ambos subárboles es que en el subárbol izquierdo se genera la variable a_i y en el derecho b_j, pero el proceso de generación es el mismo. En la figura 6.12 se muestra el subárbol final correspondiente al producto de subexpresiones.

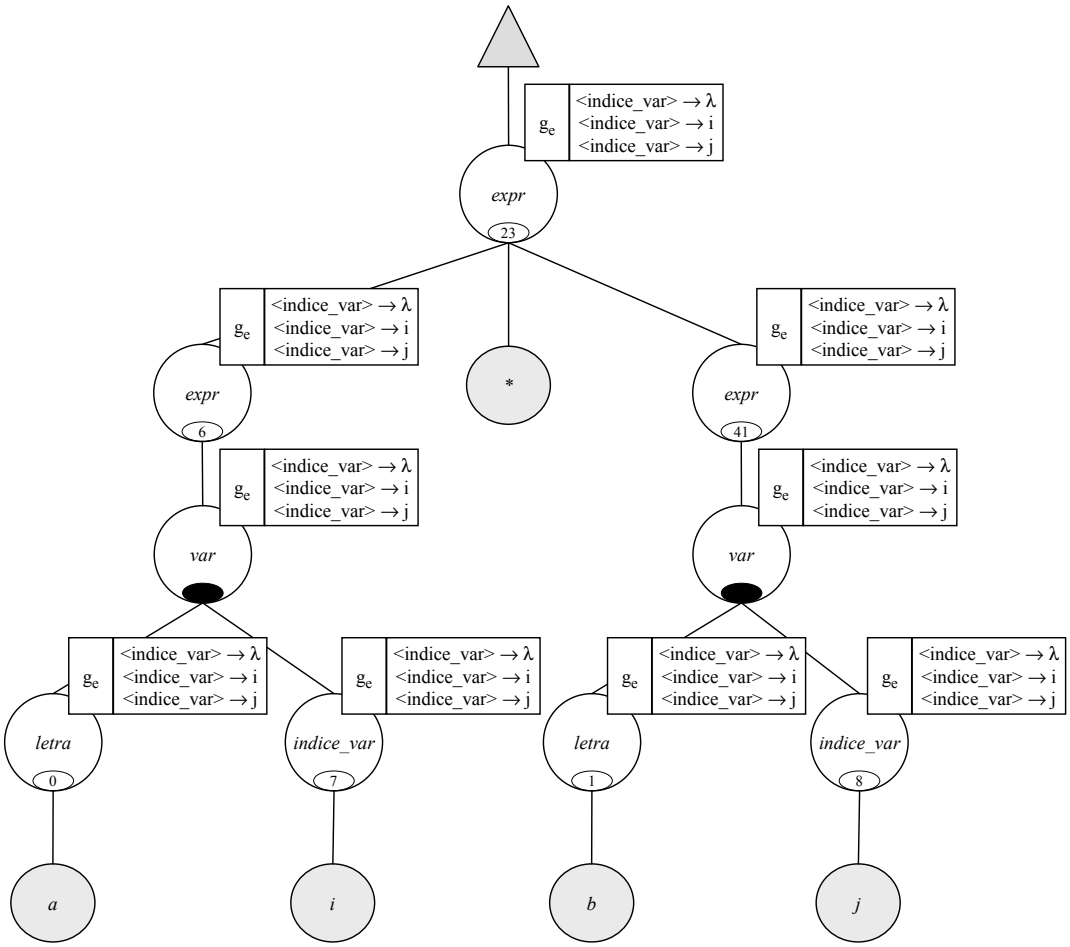

Figura 6.12: Subárbol completo de derivación de EGC a partir de la gramática G_{C_SUM}.

En el ejemplo de uso del método que se ha presentado se puede apreciar que el diseño de la gramática de Christiansen impide que se generen fenoti-

pos incorrectos. La gramática se adapta de manera que las reglas disponibles en el primer atributo de cada no terminal para su expansión son precisamente aquellas que conducen siempre a fenotipos correctos. Aunque puede ocurrir, como ya se resaltó en la descripción del método, que debido a la adaptabilidad de la gramática, un símbolo no se pueda expandir por no existir producciones para su derivación en la gramática de su primer atributo. No obstante, dada una gramática de Christianen en la que pueden ocurrir situaciones como la descrita, seguramente es posible diseñar otra equivalente que evite dichas situaciones. En el problema de regresión simbólica lógica se muestra un ejemplo. Por lo tanto, se podría decir que, en general,en EGC se evita la generación de fenotipos semánticamente incorrectos mientras que en EGA sí es posible, aunque son rechazados en el momento en el que se detecta su incorrección, y no entran a formar parte de la población. No obstante, como las gramáticas de Christiansen son extensiones de las gramáticas de atributos, siempre existe la posibilidad de plantear soluciones "combinadas". En este tipo de soluciones, algunas restricciones semánticas se implementan mediante la adaptabilidad de la gramática, y otras mediante comprobaciones sobre los atributos de la gramática.

6.2.3. Comparativa con EGA

Para resolver el ejemplo con EGA hay que diseñar una gramática de atributos que se adecúe a las restricciones semánticas del problema. Un posible diseño, partiendo de la gramática independiente del contexto G_{I_SUM}, consistiría en:

- mantener (utilizando atributos) una lista de los índices definidos y que, por lo tanto, son válidos como subíndices de variables para generar fenotipos correctos.
- inicializar la lista de índices válidos para que esté vacía.
- actualizar la lista de índices válidos cada vez que se define un índice, es decir, cada vez que se aplica una de las reglas del no terminal <*indice_sum*>.
- propagar la lista de índices válidos a través del árbol del derivación utilizando el mecanismo de herencia de atributos.
- generar un error semántico, y el correspondiente rechazo del fenotipo, si se selecciona, a partir de un codón, una regla para el no terminal <*indice_var*> que corresponda a un índice que no pertenezca a la lista de los índices definidos.

La inicialización de la lista de índices se puede realizar en la raíz del árbol de derivación, cuyo símbolo es el axioma de la gramática. Como la gramática de partida, G_{I_SUM}, tiene varias reglas para el axioma, sería necesario comprobar en cada regla la condición de raíz del árbol para realizar la inicialización. Para evitar estas múltiples comprobaciones, se modifica la gramática como habitualmente se hace en estos casos. Se añade a la gramática un nuevo axioma, con una sola regla, en cuya parte derecha está el antiguo axioma. En esta regla se incorporan las acciones específicas de la raíz del árbol.

La definición formal de la gramática de atributos sería la siguiente. Se define como $G_{A\_SUM} = \{\Sigma_T, \Sigma_N, S, P, K\}$, donde el conjunto de terminales Σ_T es:

$$\Sigma_T = \{a, b, ..., z, \sum, (,), [,], ;\}$$

Ninguno de los símbolos terminales tiene atributos definidos y por este motivo, para simplificar la notación, se omiten los paréntesis que acompañan a cada símbolo en la definición formal de una gramática.

El conjunto de los símbolos no terminales Σ_N se define como:

$$\begin{aligned}\Sigma_N = \{\quad & expresion,\\ & expr\ (ListaIndices\ \ indices),\\ & indice\_sum\ (ListaIndices\ \ indices\_previos,\ indices\_actuales),\\ & indice\_var\ (ListaIndices\ \ indices),\\ & var\ (ListaIndices\ \ indices),\\ & letra\\ \}\end{aligned}$$

El símbolo $< expresion >$ es el nuevo axioma de la gramática y no tiene atributos definidos. El tipo *ListaIndices*, al cual pertenecen todos los atributos de la gramática, es una lista que puede ser vacía, o contener elementos pertenecientes al conjunto de índices $\{i, j, k\}$. Los símbolos $<expr>$, $<indice\_var>$ y $<var>$ sólo tienen un atributo llamado *indices*, de tipo heredado, y representa la lista de índices válidos como subíndices de variables. Este atributo permite propagar la lista de índices válidos a través del árbol del derivación. El símbolo $<indice\_sum>$ tiene definidos dos atributos. El primero, *indices_previos*, es heredado, y almacena la lista de índices válidos definidos hasta el momento de la aparición del símbolo. El segundo atributo, *indices_actuales*, también de tipo *ListaIndices*, es sintetizado, y su valor es el resultado de añadir al primer atributo el nuevo

índice que aparece en la parte derecha de la regla. Por último, el símbolo <*letra*> no tiene atributos definidos.

El axioma de la gramática es:

$$S = expresion$$

El conjunto P de producciones está formado por las siguientes reglas con sus acciones semánticas:

(0) <*expresion*> → <*expr*>
{
<*expr*> .*indices* = Φ
}

(0) <*expr*> → <*expr*>$_1$ + <*expr*>$_2$
{
<*expr*>$_1$.*indices* =<*expr*> .*indices*
<*expr*>$_2$.*indices* =<*expr*> .*indices*
}
(1) <*expr*> → <*expr*>$_1$ − <*expr*>$_2$
{
<*expr*>$_1$.*indices* =<*expr*> .*indices*
<*expr*>$_2$.*indices* =<*expr*> .*indices*
}
(2) <*expr*> → <*expr*>$_1$ ∗ <*expr*>$_2$
{
<*expr*>$_1$.*indices* =<*expr*> .*indices*
<*expr*>$_2$.*indices* =<*expr*> .*indices*
}
(3) <*expr*> → <*expr*>$_1$ / <*expr*>$_2$
{
<*expr*>$_1$.*indices* =<*expr*> .*indices*
<*expr*>$_2$.*indices* =<*expr*> .*indices*
}
(4) <*expr*> → (<*expr*>$_1$)
{
<*expr*>$_1$.*indices* =<*expr*> .*indices*
}
(5) <*expr*> → $\sum$[<*indice_sum*>; <*expr*>$_1$]
{
<*indice_sum*> .*indices_previos* =<*expr*> .*indices*
<*expr*>$_1$.*indices* =<*indice_sum*> .*indices_actuales*
}
(6) <*expr*> → <*var*>
{
<*var*> .*indices* =<*expr*> .*indices*

}

(0) <var> → <letra><indice_var>
{
<indice_var> .indices =<var> .indices
}

(0) <indice_sum> → i
{
<indice_sum> .indices_actuales =<indice_sum> .indices_previos $\cup \{i\}$
}
(1) <indice_sum> → j
{
<indice_sum> .indices_actuales =<indice_sum> .indices_previos $\cup \{j\}$
}
(2) <indice_sum> → k
{
<indice_sum> .indices_actuales =<indice_sum> .indices_previos $\cup \{k\}$
}

(0) <indice_var> → λ { }
(1) <indice_var> → i
{
SI ($i \notin$<indice_var> .indices) ERROR SEMÁNTICO
}
(2) <indice_var> → j
{
SI ($j \notin$<indice_var> .indices) ERROR SEMÁNTICO
}
(3) <indice_var> → k
{
SI ($k \notin$<indice_var> .indices) ERROR SEMÁNTICO
}

(0) <letra> → a { }
(1) <letra> → b { }
(2) <letra> → c { }
(3) <letra> → d { }
... .

Por último, la gramática no tiene información global, es decir, $K = \Phi$.

No se describe el proceso de construcción de un fenotipo con la gramática G_{A_SUM} porque el objetivo de este apartado no es mostrar el algoritmo de transformación, que ya se describió en el capítulo dedicado a EGA, sino comparar EGA y EGC en el siguiente aspecto. En EGC la adaptación de la gramática impide, en general, la generación de fenotipos incorrectos semánti-

camente. Sin embargo, en EGA, los fenotipos incorrectos se detectan durante su construcción, y se rechazan como posibles integrantes de la población. El resultado final es el mismo, la población nunca contiene individuos semánticamente incorrectos, pero se consigue de distinta manera, y se podría esperar que, en general, EGC sea más eficiente que EGA en términos de tiempo. Efectivamente, en EGC no se consumen los tiempos que en EGA sí se hace en los fenotipos incorrectos, ya que, desde el comienzo de su construcción hasta que se detecta su incorrección semántica se consume un tiempo que, en general, no se consume en EGC.

6.3. Regresión simbólica de funciones lógicas

Una vez descrito el método EGC, se presenta la resolución de un problema específico, en particular, un problema similar al de regresión simbólica pero situado en el ámbito de las funciones lógicas. En los próximos apartados se plantea el problema concreto que se quiere resolver, se describe su solución con EGC y por último, se realiza una comparativa de dicha solución con tres propuestas distintas para resolver el problema con EG.

6.3.1. Planteamiento del problema

El problema propuesto para su resolución con EGC es el siguiente: dada una función lógica de un determinado número de variables, encontrar una expresión simbólica equivalente que solamente tenga operadores de uno de los siguientes conjuntos completos $\{and, or, not\}$, $\{nand\}$ o $\{nor\}$.

Aunque el conjunto de operadores lógicos $\{and, or, not, nand, nor\}$ contiene cinco subconjuntos completos $\{and, or, not\}$, $\{nand\}$, $\{nor\}$, $\{and, not\}$ y $\{or, not\}$ y cada uno de ellos permite la representación de cualquier función lógica, por razones de espacio, en el problema que se presenta sólo se centrará el interés en los tres primeros subconjuntos.

6.3.2. Solución del problema con EGC

Antes de presentar la solución con EGC al problema planteado, lo cual implica el diseño de la gramática de Christiansen adecuada, se propone una gramática independiente del contexto, $G_{I\_LOG4POST}$, para utilizarla como punto de partida. Esta gramática utiliza notación postfija (polaca inversa) con el objetivo de simplificar la programación de la función de *fitness*, ya que, parece mucho más sencillo un evaluador de expresiones en

notación postfija o prefija que en notación infija. Por otra parte, el número de variables lógicas que permite la gramática son 4. Como puede observarse, el subíndice del nombre de la gramática es explicativo, *I* significa gramática independiente del contexto, *LOG4* indica que la gramática genera expresiones con 4 variables lógicas como máximo, y *POST* hace referencia al tipo de notación, en este caso, notación postfija.

La definición formal de la gramática sería la siguiente. Se define como $G_{I\_LOG4POST} = \{\Sigma_T, \Sigma_N, S, P\}$, donde el conjunto de terminales Σ_T es:

$$\Sigma_T = \{and, or, not, nand, nor, v0, v1, v2, v3\}$$

El conjunto de los símbolos no terminales Σ_N se define como:

$$\Sigma_N = \{fb, opb, opu\}$$

donde fb representa una función lógica, *opb* un operador binario y *opu* un operador unario.

El axioma de la gramática es:

$$S = fb$$

El conjunto P de producciones está formado por las siguientes reglas:

$$\begin{array}{llll}
(0) & <fb> & \rightarrow & <fb><fb><opb> \\
(1) & & | & <fb><opu> \\
(2) & & | & v_0 \\
(3) & & | & v_1 \\
(4) & & | & v_2 \\
(5) & & | & v_3 \\
(0) & <opb> & \rightarrow & and \\
(1) & & | & or \\
(2) & & | & nand \\
(3) & & | & nor \\
 & & & \\
(0) & <opu> & \rightarrow & not
\end{array}$$

La gramática $G_{I\_LOG4POST}$ genera expresiones lógicas sin discriminar si todos los operadores de la expresión pertenecen al mismo conjunto completo. Obviamente, como se verá más adelante, se puede diseñar una gramática independiente del contexto que genere expresiones lógicas en las que los

operadores pertenezcan al mismo conjunto completo. Sin embargo, la generación de expresiones lógicas que cumplan la restricción del problema que nos ocupa es muy sencilla y directa utilizando una gramática de Christiansen.

El diseño de esta gramática de Christiansen se fundamenta en dos ideas:

1. Durante la construcción del fenotipo, la aparición de un operador desencadena la eliminación de las reglas que derivan los operadores que no pertenezcan a su mismo conjunto. Por ejemplo, la aparición del operador *nand* provoca la eliminación de las reglas $<opb> \rightarrow and$, $<opb> \rightarrow or$, $<opb> \rightarrow nor$ y $<opu> \rightarrow not$. Si *not* pertenece al conjunto de operadores eliminados, entonces, también se elimina la regla $<fb> \rightarrow <fb><opu>$.

2. Durante la construcción del fenotipo, utilizando la síntesis y herencia de atributos, se propaga por el árbol de análisis la modificación del conjunto de operadores disponibles descrito en el punto 1.

Estas dos ideas se trasladan al diseño fácilmente definiendo dos atributos de tipo gramática de Christiansen para cada no terminal. El primer atributo, heredado, es la gramática que contiene las reglas de los operadores disponibles. El segundo, sintetizado, contiene la gramática que se obtiene eliminando de la gramática heredada las producciones de los operadores que corresponda en cada caso.

Formalmente se define la gramática de Christiansen como $G_{C\_LOG4POST} = \{\Sigma_T, \Sigma_N, S, P, K\}$, donde el conjunto de terminales Σ_T es:

$$\Sigma_T = \{and, or, not, nand, nor, v0, v1, v2, v3\}$$

Como puede observarse, ninguno de los símbolos terminales tiene atributos definidos.

El conjunto de los símbolos no terminales Σ_N es define como:

$$\begin{aligned} \Sigma_N = \{ \quad & fb\ (GC\ \ g_h, GC\ \ g_s), \\ & opb\ (GC\ \ g_h, GC\ \ g_s), \\ & opu\ (GC\ \ g_h, GC\ \ g_s)\} \end{aligned}$$

Como es habitual, las siglas GC significan Gramática de Christiansen. Los atributos referidos como g_h representan gramáticas heredadas, y los referidos como g_s gramáticas sintetizadas.

El axioma de la gramática es:

$$S = fb$$

El conjunto P de producciones está formado por las siguientes reglas con sus acciones semánticas:

(0) $<fb> \rightarrow <fb>_1 <fb>_2 <opb>$
{
$<fb>_1 .g_h = <fb> .g_h$
$<fb>_2 .g_h = <fb>_1 .g_s$
$<opb> .g_h = <fb>_2 .g_s$
$<fb> .g_s = <opb> .g_s$
}

(1) $<fb> \rightarrow <fb>_1 <opu>$
{
$<fb>_1 .g_h = <fb> .g_h$
$<opu> .g_h = <fb>_1 .g_s$
$<fb> .g_s = <opu> .g_s$
}

(2) $<fb> \rightarrow v_0$
{
$<fb> .g_s = <fb> .g_h$
}

(3) $<fb> \rightarrow v_1$
{
$<fb> .g_s = <fb> .g_h$
}

(4) $<fb> \rightarrow v_2$
{
$<fb> .g_s = <fb> .g_h$
}

(5) $<fb> \rightarrow v_3$
{
$<fb> .g_s = <fb> .g_h$
}

(0) $<opb> \rightarrow and$
{
$<opb> .g_s = <opb> .g_h - \{<opb> \rightarrow nand\ ,\ <opb> \rightarrow nor\}$
}

(1) $<opb> \rightarrow or$
{

$<opb>.g_s = <opb>.g_h - \{<opb> \rightarrow nand\ ,\ <opb> \rightarrow nor\}$
}

(2) $<opb> \rightarrow nand$
{
$<opb>.g_s = <opb>.g_h - \{<opb> \rightarrow and\ ,\ <opb> \rightarrow or$
$<opb> \rightarrow nor\ ,\ <opu> \rightarrow not$
$<fb> \rightarrow <fb><opu>\}$
}

(3) $<opb> \rightarrow nor$
{
$<opb>.g_s = <opb>.g_h - \{<opb> \rightarrow and\ ,\ <opb> \rightarrow or$
$<opb> \rightarrow nand\ ,\ <opu> \rightarrow not$
$<fb> \rightarrow <fb><opu>\}$
}

(0) $<opu> \rightarrow not$
{
$<opu>.g_s = <opu>.g_h - \{<opb> \rightarrow nand\ ,\ <opb> \rightarrow nor\}$
}

Por último, la gramática no tiene información global, es decir, $K = \Phi$.

Como se puede observar en las producciones de la gramática, la regla (0) del no terminal $<fb>$ representa la transformación de una función lógica en una operación lógica binaria cuyos operandos son también funciones lógicas. En la acción semántica de esta regla, la gramática heredada por el símbolo de la parte izquierda en su primer atributo, es heredada por su primer hijo. Las posibles modificaciones relativas a los operadores en el subárbol de este primer hijo, se sintetizan en su segundo atributo, que es heredado por su hermano (el segundo hijo) en su primer atributo. De nuevo, los cambios en los operadores en el subárbol del segundo hijo, son sintetizados en su segundo atributo, que es heredado por su hermano derecho (el tercer hijo) que corresponde al operador binario. La derivación del operador binario elimina de su gramática heredada los operadores que no pertenecen a su mismo conjunto completo y la gramática resultante pasa a ser el segundo atributo sintetizado de su padre.

En la regla (1) del no terminal $<fb>$, que representa la transformación de una función lógica en una operación lógica unaria, la dinámica de herencia y síntesis de gramáticas es similar a la de la regla (0).

En cuanto a las producciones de los operadores lógicos, todas siguen el mismo esquema. El primer atributo corresponde a la gramática heredada que contiene los operadores disponibles. El segundo, es la gramática que se

obtiene eliminando de la heredada las producciones de los operadores que no pertenecen al mismo conjunto completo.

La gramática $G_{C\_LOG4POST}$ tiene un diseño que permite que se generen situaciones que podrían considerarse anómalas. En efecto, puede ocurrir que en la expansión de un símbolo, la gramática con la que se tiene que derivar (la almacenada en su primer atributo), no contenga ninguna regla para el propio símbolo. En la sección 6.2.1 se ha mencionado que, para gestionar estas situaciones, en EGC se propone el siguiente mecanismo: se añade una regla 0) al algoritmo de transformación de genotipo a fenotipo de EGA que establece que, antes de hacer la expansión de un nodo, se comprueba si existen producciones para su derivación, y en el caso de no existir, se rechaza el fenotipo en construcción igual que se hace en EGA.

Para ejemplificar la situación descrita, se muestran los primeros pasos del algoritmo de generación de fenotipos, a partir del siguiente genotipo:

7	6	8	9	2	47	42	6	8	14	8

En el primer paso del algoritmo se construye la raíz del árbol con el axioma de la gramática. El primer atributo del axioma se inicializa con la gramática $G_{C\_LOG4POST}$ que, en su estado inicial, contiene producciones para todos los operadores y para la operación unaria. En la figura 6.13 se muestra el árbol de derivación después del primer paso del algoritmo. Puede observarse que se mantiene la representación habitual de los árboles. Recuérdese que se simplifica la información que aparece en el árbol asociada a cada atributo. En concreto, en los atributos de tipo gramática de Christiansen, no aparecen todas las producciones de la gramática sino, del conjunto de las producciones que pueden estar o no presentes, aquellas que sí los están. En particular, en la gramática $G_{C\_LOG4POST}$, las producciones que pueden estar o no presentes son las de los operadores y la que representa una operación unaria.

El proceso continúa con el primer codón, 7, que selecciona la regla (1) del axioma para su expansión. En la figura 6.14 se muestra el resultado de la expansión y se resalta la herencia de atributos con una flecha punteada. En concreto, el primer hijo hereda en su primer atributo la gramática almacenada en el primer atributo de su padre.

El siguiente codón vale 6, y el nodo más profundo y situado más a la izquierda corresponde al símbolo $<fb>$. Por lo tanto, la regla para la expansión es la (0), que corresponde a la operación binaria de dos subfunciones lógicas. El árbol resultante se muestra en la figura 6.15. Se puede observar

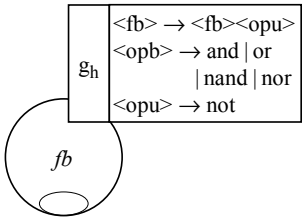

Figura 6.13: Árbol de derivación de EGC a partir de la gramática $G_{C\_LOG4POST}$ después de la construcción del nodo raíz.

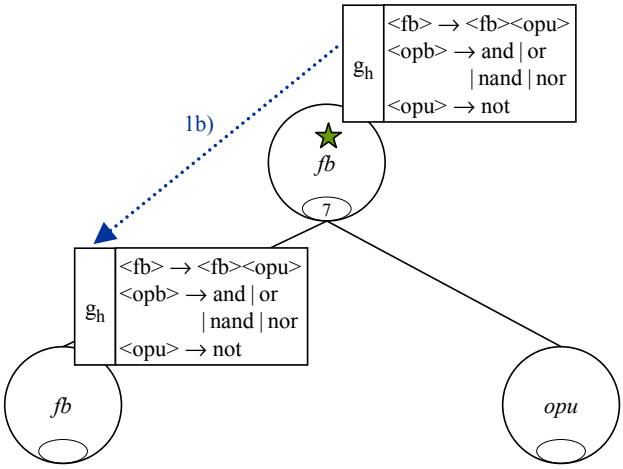

Figura 6.14: Árbol de derivación de EGC a partir de la gramática $G_{C\_LOG4POST}$ después de la expansión del nodo raíz.

que el primer hijo hereda en su primer atributo la gramática almacenada en el primer atributo de su nodo padre, en la que todavía hay producciones para todos los operadores lógicos y la operación unaria.

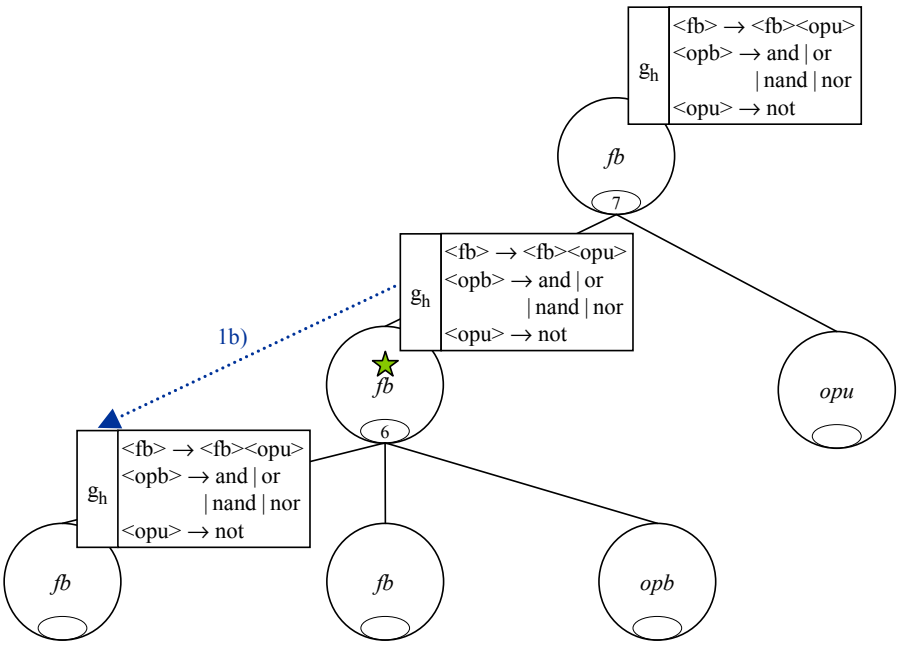

Figura 6.15: Árbol de derivación de EGC a partir de la gramática $G_{C\_LOG4POST}$ después de la expansión de la segunda aparición del símbolo $<fb>$.

El siguiente nodo a expandir también corresponde al símbolo $<fb>$ y el codón actual es 8. Por lo tanto, se utiliza la regla (2) del no terminal, la cual, en su parte derecha, tiene un único símbolo terminal, el v_0. En la figura 6.16 se muestra el árbol de derivación resultante. El nodo que se acaba de expandir, marcado en la figura con una estrella, sólo tiene un hijo, que además es un terminal, por lo tanto, se aplica la regla 2b) relativa al cálculo de los atributos sintetizados, si los hubiere. En este caso sí hay atributos sintetizados, en concreto, el segundo atributo del símbolo $<fb>$, que es una gramática de Christiansen, se evalúa como copia del primer atributo. En la figura 6.16 queda reflejada esta evaluación de atributos con una flecha sólida. Por otra parte, el nodo que se acaba de expandir se considera terminado, y como tiene nodos hermanos, se aplica la regla 2a) relativa a la evaluación de atributos heredados entre hermanos. En concreto su primer hermano derecho

hereda en su primer atributo la gramática sintetizada en el segundo atributo, como se indica en la figura 6.16 con una flecha punteada.

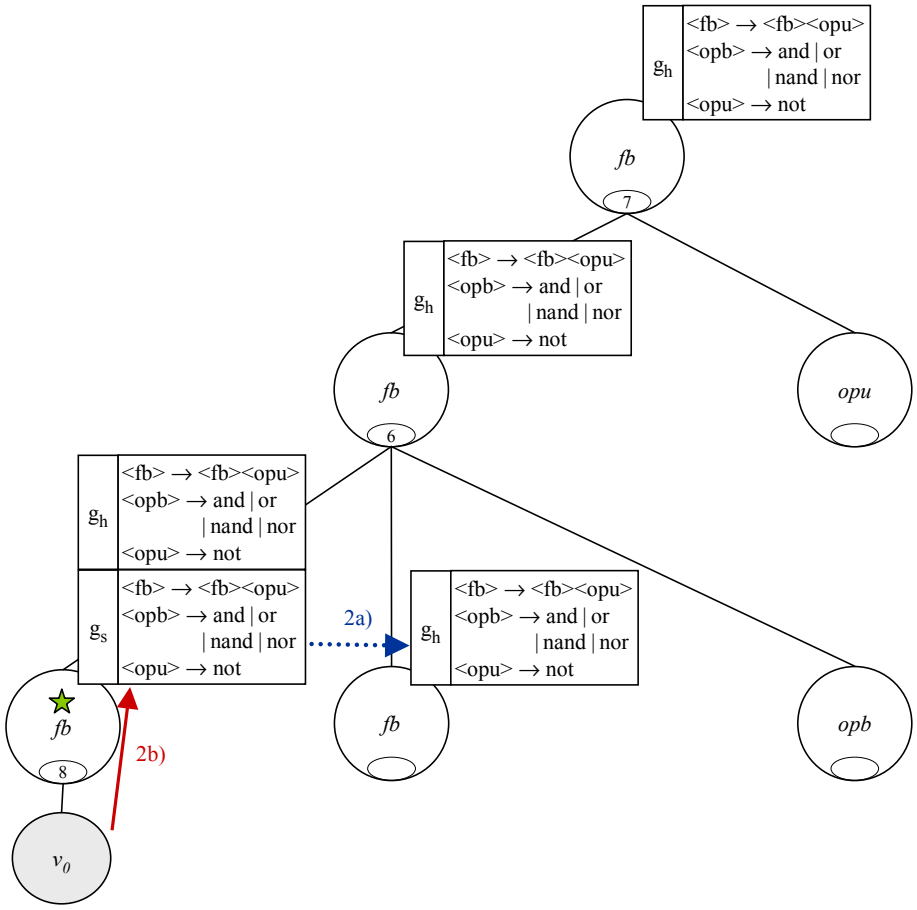

Figura 6.16: Árbol de derivación de EGC a partir de la gramática $G_{C\_LOG4POST}$, después de la expansión de la tercera aparición del símbolo $<fb>$.

El proceso continúa con el codón 9 y el siguiente símbolo a expandir es de nuevo $<fb>$. La regla que se selecciona para la expansión es la (3). Esta regla tiene en su parte derecha un único símbolo, el terminal v_1. La evaluación de atributos sintetizados y heredados desencadenada por esta última expansión es similar a la evaluación que se describió en la anterior operación de expansión. El árbol resultante se muestra en la figura 6.17.

El siguiente nodo a expandir es el correspondiente al no terminal $<opb>$

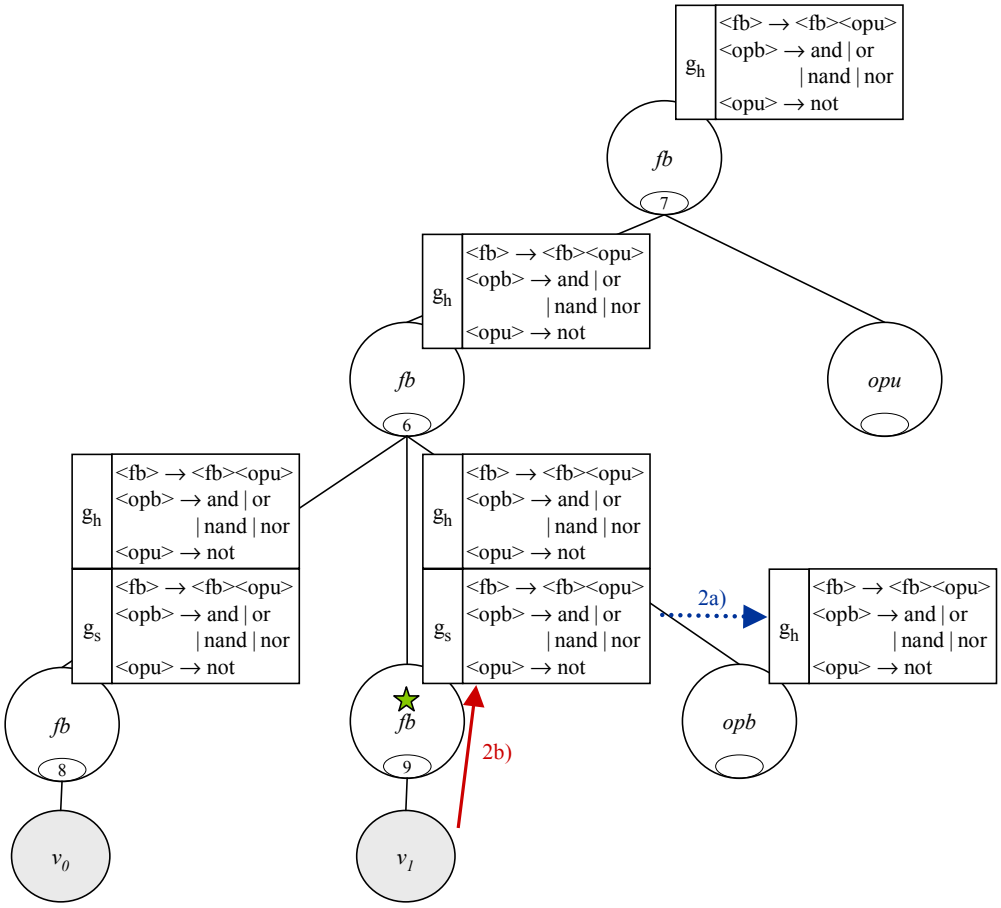

Figura 6.17: Árbol de derivación de EGC a partir de la gramática $G_{C\_LOG4POST}$ después de la expansión de la cuarta aparición del símbolo $<fb>$.

y el codón actual vale 2. La regla que se utiliza para la expansión es la (2) y corresponde al operador *nand*. En la acción semántica de esta regla se especifica que la gramática sintetizada que se almacena en el segundo atributo de <*opb*> se obtiene eliminando un conjunto de reglas de la gramática heredada por el símbolo y almacenada en su primer atributo. En concreto, las reglas que se eliminan son las de los operadores *and*, *or*, *nor* y *not*, y la regla correspondiente a la operación unaria. En la figura 6.18 se resalta con una flecha sólida el proceso de síntesis descrito. De esta manera, el segundo atributo del símbolo <*opb*> contiene una instancia de la gramática $G_{C\_LOG4POST}$ en la que no existe ninguna regla para la operación unaria, y solamente existe una regla para los operadores, la correspondiente a *nand*.

En la figura 6.18 se puede observar que la expansión del último nodo, al ser éste el último hijo de su nodo padre, el correspondiente a la segunda aparición del símbolo <fb>, desencadena el cálculo de atributos sintetizados de éste símbolo. En la figura 6.18 se representa dicha síntesis con una flecha sólida. La siguiente acción del algoritmo correspondería al cálculo de atributos heredados entre hermanos, ya que, la segunda aparición del símbolo <fb> tiene a su derecha un nodo hermano, el del símbolo <*opu*>. La gramática especifica que el primer atributo del símbolo <*opu*> se evalúa como una copia del segundo atributo del símbolo <fb>, y en la figura 6.18 se representa con una flecha punteada.

El siguiente nodo a expandir es el del símbolo <*opu*>, pero la gramática almacenada en su primer atributo, que es la que se utiliza para su derivación, no contiene reglas para él. Esta situación es precisamente la que se quería ejemplificar con el genotipo que se está estudiando. En principio, no sería un problema, ya que la regla 0) del algoritmo de transformación de genotipo a fenotipo de EGC establece el mecanismo de actuación ante tal circunstancia. Antes de expandir un nodo, se comprueba si la gramática almacenada en su primer atributo contiene reglas para su expansión, y si no es así, se interrumpe el proceso de generación del fenotipo y se rechaza. No obstante, aunque el algoritmo funciona adecuadamente, existe un consumo de tiempo no productivo en los fenotipos que finalmente son rechazados.

Desde el punto de vista de la minimización del tiempo de ejecución, parece razonable diseñar gramáticas de Christiansen en las que no ocurran situaciones como la descrita. Sería interesante disponer de una técnica que permitiera comprobar si una gramática de Christiansen está bien diseñada en ese aspecto. Queda fuera del alcance de esta tesis el posible desarrollo de una técnica de comprobación de gramáticas de Christiasen y se deja como una línea de investigación abierta.

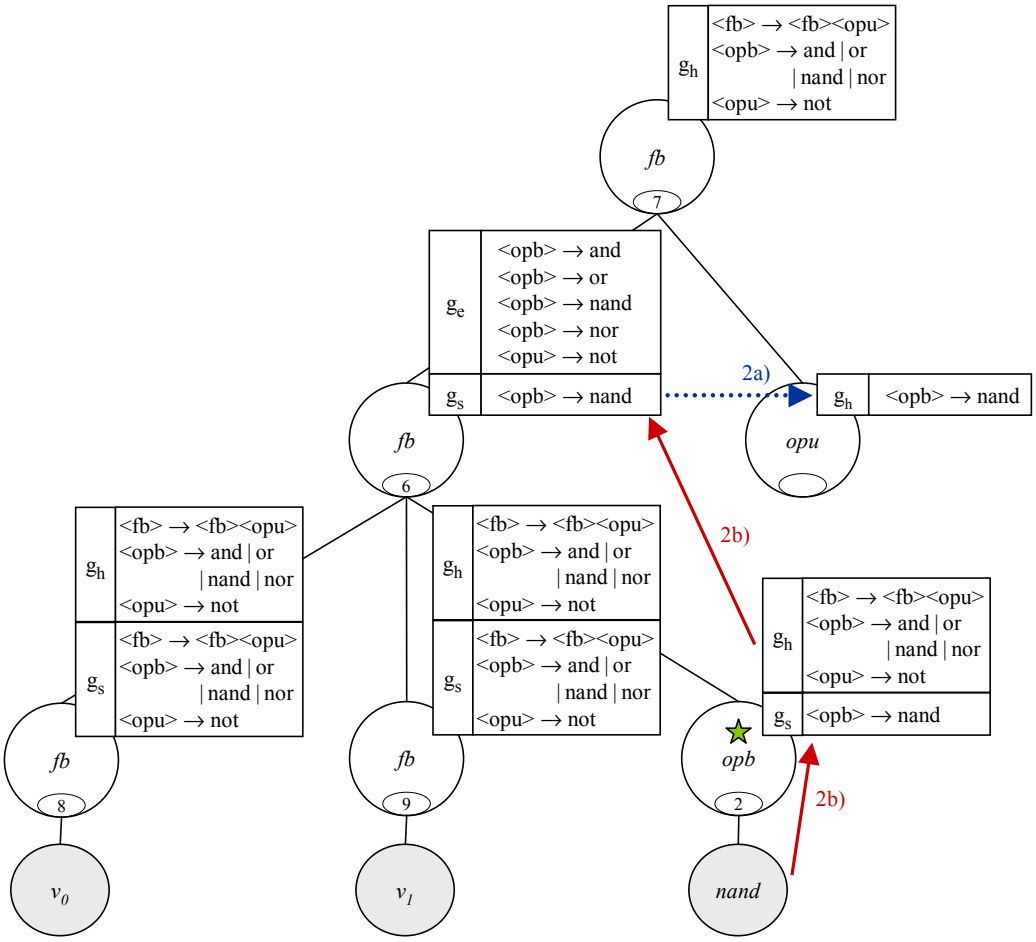

Figura 6.18: Árbol de derivación de EGC a partir de la gramática $G_{C\_LOG4POST}$ después de la expansión de la primera aparición del símbolo *<opb>*.

Se ha realizado un experimento con el objetivo de comparar el tiempo que se tarda en resolver con EGC el mismo problema de regresión simbólica lógica utilizando dos gramáticas de Christiansen distintas. La primera gramática es precisamente $G_{C\_LOG4POST}$, que como se ha demostrado, en ciertas circunstancias rechaza fenotipos en construcción con su correspondiente pérdida de tiempo. La segunda, $G_{C\_LOG4PRE}$, implementa la notación prefija, y parece que en ningún caso rechaza individuos. Las dos gramáticas comparten los conjuntos de símbolos terminales y no terminales, y también el sistema de atributos. La diferencia entre ambas está en las producciones junto con sus acciones semánticas. En concreto, las reglas de $G_{C\_LOG4PRE}$ son las siguientes:

(0) $<fb> \rightarrow <opb><fb>_1<fb>_2$
{
$<opb>.g_h = <fb>.g_h$
$<fb>_1.g_h = <opb>.g_s$
$<fb>_2.g_h = <fb>_1.g_s$
$<fb>.g_s = <fb>_2.g_s$
}

(1) $<fb> \rightarrow <opu><fb>_1$
{
$<opu>.g_h = <fb>.g_h$
$<fb>_1.g_h = <opu>.g_s$
$<fb>.g_s = <fb>_1.g_s$
}

(2) $<fb> \rightarrow v_0$
{
$<fb>.g_s = <fb>.g_h$
}

(3) $<fb> \rightarrow v_1$
{
$<fb>.g_s = <fb>.g_h$
}

(4) $<fb> \rightarrow v_2$
{
$<fb>.g_s = <fb>.g_h$
}

(5) $<fb> \rightarrow v_3$
{
$<fb>.g_s = <fb>.g_h$
}

(0) $<opb> \rightarrow and$

$$
\begin{array}{ll}
 & \{ \\
 & \quad <opb> .g_s = <opb> .g_h - \{<opb> \rightarrow nand\ ,\ <opb> \rightarrow nor\} \\
 & \} \\
(1) & <opb> \quad \rightarrow \quad or \\
 & \{ \\
 & \quad <opb> .g_s = <opb> .g_h - \{<opb> \rightarrow nand\ ,\ <opb> \rightarrow nor\} \\
 & \} \\
(2) & <opb> \quad \rightarrow \quad nand \\
 & \{ \\
 & \quad <opb> .g_s = <opb> .g_h - \{<opb> \rightarrow and\ ,\ <opb> \rightarrow or \\
 & \qquad\qquad\qquad\qquad\qquad\quad <opb> \rightarrow nor\ ,\ <opu> \rightarrow not \\
 & \qquad\qquad\qquad\qquad\qquad\quad <fb> \rightarrow <opu><fb>\} \\
 & \} \\
(3) & <opb> \quad \rightarrow \quad nor \\
 & \{ \\
 & \quad <opb> .g_s = <opb> .g_h - \{<opb> \rightarrow and\ ,\ <opb> \rightarrow or \\
 & \qquad\qquad\qquad\qquad\qquad\quad <opb> \rightarrow nand\ ,\ <opu> \rightarrow not \\
 & \qquad\qquad\qquad\qquad\qquad\quad <fb> \rightarrow <opu><fb>\} \\
 & \}
\end{array}
$$

$$
\begin{array}{ll}
(0) & <opu> \quad \rightarrow \quad not \\
 & \{ \\
 & \quad <opu> .g_s = <opu> .g_h - \{<opb> \rightarrow nand\ ,\ <opb> \rightarrow nor\} \\
 & \}
\end{array}
$$

Utilizando las dos gramáticas, $G_{C\_LOG4POST}$ y $G_{C\_LOG4PRE}$, se ha resuelto el problema de regresión simbólica lógica, con la función objetivo de cuatro variables (v_0 *or* v_1) *and* (v_2 *or* v_3). Para cada gramática se han realizado 200 experimentos independientes con los parámetros de ejecución que se muestran en la tabla 6.6.

La función de *fitness* debe asignar el mayor valor al mejor individuo. Aunque el *raw fitness* definido como el número de puntos en los cuales la función lógica candidata coincide con la función objetivo cumple el requisito, se ha utilizado el *fitness* ajustado de acuerdo con la definición de [Koza (1992)], que también tiene la propiedad requerida, y además, distingue más finamente individuos próximos a la solución. La fórmula para calcular el *fitness* ajustado $a(i,t)$ de un individuo i en un instante de tiempo t en función de su *fitness* estándar, $s(i,t)$, es la siguiente:

$$a(i,t) = \frac{1}{1 + s(i,t)}$$

El *fitness* estándar se define en función del *raw fitness*, $r(i,t)$, de la siguiente manera:

Tamaño de la población	500
Número máximo de generaciones	400
Número mínimo de codones (inicial)	10
Número máximo de codones (inicial)	50
Valor mínimo de codón	0
Valor máximo de codón	256
Operador de *wrapping*	inhabilitado
Elitismo	NO
Tasa de recombinación	90 %
Tasa de mutación	90 %

Tabla 6.6: Parámetros de ejecución del algoritmo EGC para el problema de regresión lógica con la función objetivo $(v_0\ or\ v_1)\ and\ (v_2\ or\ v_3)$ y las gramáticas de Christiansen $G_{C\_LOG4POST}$ y $G_{C\_LOG4PRE}$.

$$s(i,t) = 16 - r(i,t)$$

Definido de esta manera, el *fitness* ajustado varía desde el valor 1 asignado al mejor individuo hasta 1/17 correspondiente el peor individuo.

Los resultados obtenidos confirman la hipótesis acerca del mejor rendimiento de la gramática $G_{C\_LOG4PRE}$ en cuanto a tiempo de proceso. Con la gramática $G_{C\_LOG4PRE}$ se ejecutaron los 200 experimentos en 18 minutos, y con $G_{C\_LOG4POST}$ el tiempo subió a 26 minutos.

En cuanto al rendimiento en términos de soluciones encontradas, con ambas gramáticas se obtienen rendimientos similares como se puede observar en la figura 6.19.

6.3.3. Comparativa con EG

No es fácil encontrar un problema en el que se puedan comparar, en términos de rendimiento, los resultados obtenidos cuando se utilizan los dos métodos de resolución EG y EGC. Por una parte, si las restricciones del problema son completamente expresables mediante una gramática independiente del contexto, parece razonable pensar que EGC es un método cuya capacidad expresiva excede a las necesidades del problema. En este caso sería suficiente conseguir que EGC no empeore los resultados obtenidos por EG. Por otra parte, si el problema tiene unas restricciones que no se pueden expresar únicamente con una gramática independiente del contexto, entonces,

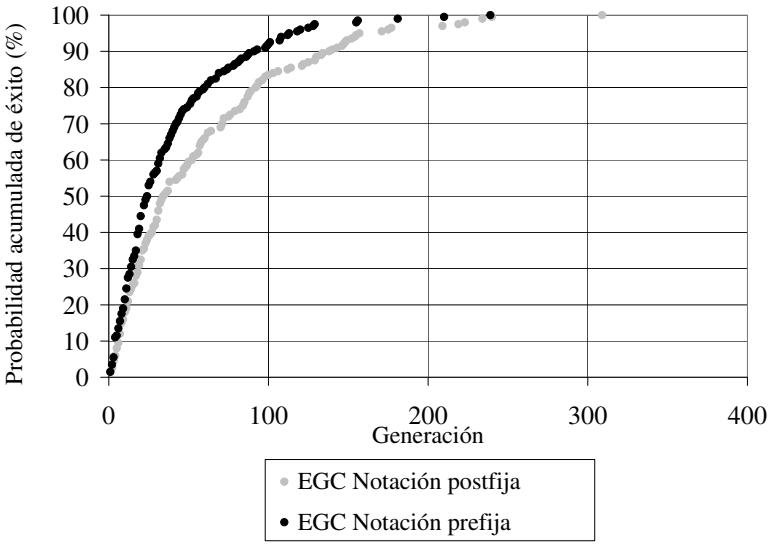

Figura 6.19: Comparativa de las probabilidades acumuladas de éxito de EGC para el problema de regresión lógica con la función objetivo (v_0 *or* v_1) *and* (v_2 *or* v_3), y las gramáticas de Christiansen $G_{C\_LOG4POST}$ y $G_{C\_LOG4PRE}$.

simplemente, no se puede abordar de forma natural con EG aunque, en algunos casos, sea posible incorporar ciertas restricciones a través de la función de *fitness*.

En los casos en los que la capacidad expresiva de EGC excede a las necesidades del problema, se puede justificar su uso, en lugar de EG, por el confort que proporciona en el diseño de soluciones.

La elección del problema de regresión lógica que nos ocupa no ha sido casual. Precisamente se ha elegido para poder comparar la solución que proporciona EG y la que se obtiene con EGC, ya que, la restricción que impone el problema acerca de la pertenencia al mismo conjunto completo de todos los operadores del fenotipo, se puede expresar con una gramática independiente del contexto.

En este apartado se describen tres propuestas para resolver el problema de regresión lógica con EG y se comparan sus resultados con EGC. La función objetivo sigue siendo la función $(v_0\ or\ v_1)\ and\ (v_2\ or\ v_3)$, y la solución al problema con EGC, con la que se compararan los resultados, corresponde a la gramática en notación prefija $G_{C\_LOG4PRE}$, cuyos datos de rendimiento se muestran en la figura 6.19. En cuanto a los parámetros configurables del método (tamaño de la población, máximo número de generaciones, etc) se utilizan los mismos en las tres propuestas, y coinciden con los que se utilizaron con EGC (se pueden consultar en la tabla 6.6). Por último, en los tres casos se realizan 200 ejecuciones independientes para obtener las curvas de rendimiento.

En primer lugar se resuelve el problema con la gramática independiente del contexto definida como $G_{I\_LOG4PRE} = \{\Sigma_T, \Sigma_N, S, P\}$, donde el conjunto de terminales Σ_T es:

$$\Sigma_T = \{and, or, not, nand, nor, v0, v1, v2, v3\}$$

El conjunto de los símbolos no terminales Σ_N se define como:

$$\Sigma_N = \{fb, opb, opu\}$$

El axioma de la gramática es:

$$S = fb$$

El conjunto P de producciones está formado por las siguientes reglas:

$$\begin{array}{llll} (0) & <fb> & \rightarrow & <opb><fb><fb> \\ (1) & & | & <opu><fb> \\ (2) & & | & v_0 \end{array}$$

(3)		|	v_1
(4)		|	v_2
(5)		|	v_3
(0)	<*opb*>	→	*and*
(1)		|	*or*
(2)		|	*nand*
(3)		|	*nor*
(0)	<*opu*>	→	*not*

Obviamente, la gramática $G_{I\_LOG4PRE}$ genera expresiones lógicas sin discriminar si todos los operadores de la expresión pertenecen al mismo conjunto completo. Por lo tanto, es necesario incorporar dicha restricción de alguna manera externa a la gramática independiente del contexto. Una primera aproximación, que corresponde a la primera de las tres propuestas de solución con EG, es eliminar manualmente, de las soluciones encontradas en las 200 ejecuciones, aquellas en las que aparecen operadores que pertenecen a distintos conjuntos. Esta alternativa proporciona un rendimiento muy bajo en comparación al obtenido con EGC como se puede observar en la figura 6.20. Este rendimiento tan bajo es razonable, ya que, durante el proceso evolutivo, no se incorpora ninguna información acerca de la idoneidad de las soluciones (en cuanto a mezcla de operadores).

La segunda propuesta consiste en incorporar en la función de *fitness* la restricción relativa a la pertenencia de los operadores al mismo conjunto. Los fenotipos en los que se detecta mezcla de operadores son penalizados con el peor *fitness*. En este caso, sí existe información durante el proceso evolutivo acerca de la idoneidad de las soluciones, pero como se puede observar en la figura 6.20 el rendimiento es similar al obtenido en la primera propuesta.

Por último, la tercera propuesta utiliza una nueva gramática que genera fenotipos con operadores pertenecientes al mismo conjunto completo. Esta gramática se define como $G_{I\_LOG4DIS} = \{\Sigma_T, \Sigma_N, S, P\}$, donde el conjunto de terminales Σ_T es:

$$\Sigma_T = \{and, or, not, nand, nor, v0, v1, v2, v3\}$$

El conjunto de los símbolos no terminales Σ_N se define como:

$$\Sigma_N = \{fb, fb1, fb2, fb3, opb1, opb2, opb3, opu1\}$$

El axioma de la gramática es:

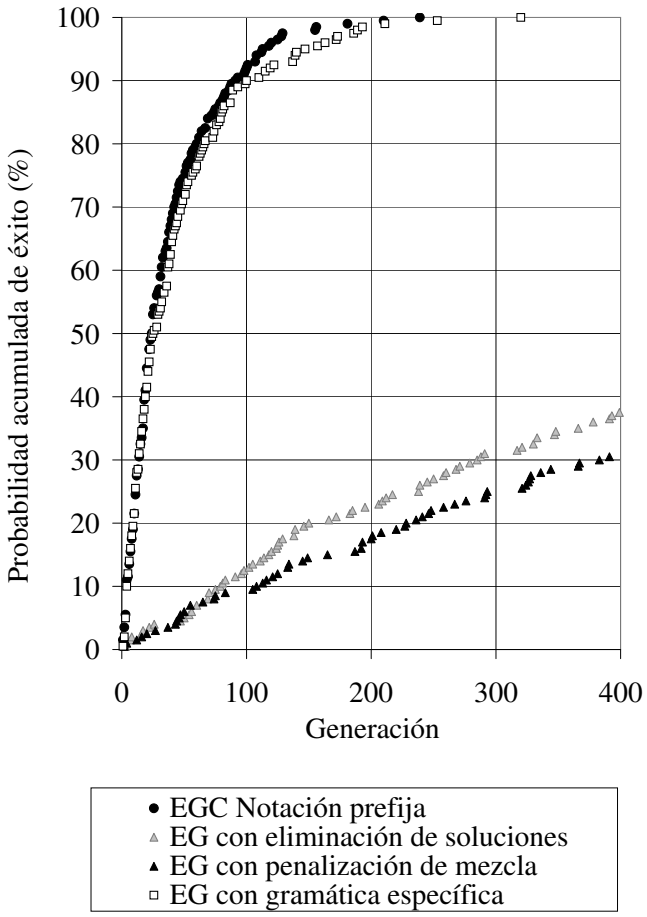

Figura 6.20: Comparativa de las probabilidades acumuladas de éxito de EG y EGC para el problema de regresión lógica con la función objetivo $(v_0\ or\ v_1)\ and\ (v_2\ or\ v_3)$.

$$S = fb$$

El conjunto P de producciones está formado por las siguientes reglas:

(0) $<fb>$ → $<fb1>|<fb2>|<fb3>$

(0) $<fb1>$ → $<opb1><fb1><fb1>$
(1) | $<opu1><fb1>$
(2) | v_0
(3) | v_1
(4) | v_2
(5) | v_3

(0) $<opb1>$ → and
(1) | or

(0) $<opu1>$ → not

(0) $<fb2>$ → $<opb2><fb2><fb2>$
(1) | v_0
(2) | v_1
(3) | v_2
(4) | v_3

(0) $<opb2>$ → $nand$

(0) $<fb3>$ → $<opb3><fb3><fb3>$
(1) | v_0
(2) | v_1
(3) | v_2
(4) | v_3

(0) $<opb3>$ → nor

El rendimiento obtenido con la nueva gramática $G_{I\_LOG4DIS}$ es similar al obtenido con EGC, como se puede observar en la figura 6.20. Es razonable esta similitud ya que en ambos métodos los fenotipos que se generan cumplen la restricción relativa a la mezcla de operadores, aunque el mecanismo para obtener dichos fenotipos es diferente. No obstante, la gramática de Christiansen es más reducida que la gramática independiente del contexto, y proporciona un diseño más compacto y directo de la solución del problema.

Es obvio que si existiera un número superior de operadores lógicos y de conjuntos completos, se incrementaría notablemente el número de producciones de la gramática independiente del contexto mientras que, la gramática de Christiansen se mantendría reducida. Esta característica, formaría parte de lo que se denomina, en el contexto de esta tesis, "confort de diseño".

Se han medido los tiempos de ejecución de los 200 experimentos realizados con EGC y los 200 realizados en cada una de las tres variantes de EG y se han obtenido los siguientes valores:

- El método EGC ha tardado 18 minutos utilizando la gramática en notación prefija. Anteriormente ya se comentó este resultado en comparación con el tiempo invertido por EGC si se utiliza la notación postfija.
- El método EG sin información acerca de las restricciones de los operadores durante el proceso evolutivo, ha tardado 65 minutos.
- El método EG que penaliza en el *fitness* a los individuos con mezcla de operadores ha tardado 84 minutos.
- El método EG que utiliza una gramática independiente del contexto que genera individuos que cumplen las restricciones de los operadores, ha tardado 12 minutos.

Los tiempos medidos parecen razonables. Los valores altos en los dos primeros escenarios de EG se pueden explicar porque muchas de las soluciones se obtienen en generaciones altas, lo que significa procesos evolutivos más largos, y por lo tanto, más costosos en tiempo. El parecido entre los tiempos de EGC y la última variante de EG, se explica de la misma manera que se explicaba su similitud en términos de rendimiento: ambos generan fenotipos correctos y comparten la misma función de *fitness*. El tiempo superior de EGC se puede explicar porque en este caso, se añade al proceso de derivación la gestión de la gramática (su adaptabilidad) mediante la propagación de valores de ciertos atributos.

Capítulo 7

GEEMMA

En el contexto de esta tesis se ha desarrollado la aplicación **GEEMMA** con el objetivo de disponer de un entorno en el que probar los métodos teóricos propuestos y así poder validar su funcionamiento y evaluar su rendimiento.

En este capítulo se describen los aspectos de **GEEMMA** más relevantes para el usuario de la herramienta, obviando todos los aspectos del diseño interno de la aplicación, excepto aquellos que son imprescindibles para el usuario.

7.1. Descripción general

GEEMMA ha sido desarrollada en Java por razones de portabilidad, sin tener en cuenta hasta la fecha el factor "velocidad de ejecución". Precisamente, uno de los futuros trabajos planificados para llevar a cabo a la finalización de esta tesis, es el desarrollo de una herramienta similar en cuanto a aspectos de funcionalidad, pero con un diseño que minimice la velocidad de ejecución de los experimentos.

GEEMMA permite realizar experimentos con cualquiera de los siguientes métodos:

- Evolución Gramatical
- Evolución Gramatical con Gramáticas de Atributos
- Evolución Gramatical con Gramáticas de Christiansen

Los tres métodos anteriores comparten el mismo algoritmo evolutivo que se describe en el apartado 7.2.

Para la realización de cualquier experimento el usuario debe proporcionar el código Java que implementa la función de *fitness* y el código que implementa la semántica. En el apartado 7.3 se describe la manera en la que el usuario puede incluir en la aplicación su propio código para.

Además del código de usuario, la realización de un experimento requiere su configuración previa en la que se establecen parámetros como por ejemplo: método (uno de los tres disponibles), tamaño de la población, tasas de los operadores genéticos, gramática independiente del contexto, etc. En el apartado 7.4 se puede consultar la lista completa de los parámetros de configuración de un experimento.

Los experimentos se pueden realizar de manera individual o bien repetir el experimento un determinado número de veces. En esta última modalidad, denominada en el entorno "batería de experimentos" la aplicación genera un fichero que contiene el informe de los resultados de cada uno de los experimentos individuales lo que permite, por ejemplo, realizar estimaciones acerca del rendimiento de los métodos, hacer estadísticas sobre la longitud de los genotipos de los mejores individuos, estudiar los fenotipos de los mejores individuos, etc. Todos los detalles relativos a la realización de experimentos están descritos en el apartado 7.4.

Además, la aplicación incorpora una componente de gestión de gramáticas independientes del contexto que permite al usuario definir cómodamente dichas gramáticas. En el apartado 7.5

7.2. Algoritmo evolutivo

El algoritmo evolutivo implementado en **GEEMMA** tiene las características que se describen a continuación.

En primer lugar, en la inicialización aleatoria de la población, la construcción de cada individuo comienza con la elección aleatoria de la longitud de su genotipo. Esta longitud, expresada en número de codones, siempre está dentro del rango establecido por el usuario en la configuración del experimento. Posteriormente, y también de manera aleatoria, se generan los codones que forman el genotipo, manteniendo los valores de los codones dentro del rango de valores prefijado por el usuario en la fase de configuración. Una vez construido el genotipo, el fenotipo del individuo se obtiene aplicando el algoritmo de transformación adecuado al método seleccionado (Evolución Gramatical o Evolución Gramatical con semántica).

En cuanto a la selección de los individuos, se realiza de manera proporcional al *fitness*.

El operador de recombinación es de un punto, y siempre se generan dos individuos hijos a partir de dos padres, salvo en los métodos que incorporan semántica, en los que solamente los hijos válidos semánticamente son incorporados a la población. El operador de mutación se aplica posteriormente a la recombinación, y opera a nivel de codón. En el capítulo 4 se puede encontrar una descripción más detallada del funcionamiento de los operadores genéticos.

Por otra parte, el reemplazo de la población es generacional con elitismo opcional. Esto significa que de una generación a la siguiente se reemplazan todos los individuos por ser generacional, y el elitismo, mantiene el mejor individuo entre generaciones. El usuario configura el uso o desuso del elitismo en la configuración del experimento.

El proceso evolutivo termina cuando es cierta alguna de las siguientes condiciones:

- Se ha encontrado una solución aceptable, es decir, un individuo cuya función de *fitness* es superior al umbral previamente establecido en la configuración del experimento.
- Se ha alcanzado el número máximo de generaciones fijado por el usuario en la configuración.

7.3. Código de usuario

El usuario de **GEEMMA** tiene la responsabilidad de proporcionar a la aplicación el código Java que implementa la función de *fitness* y la semántica definida para el experimento. El mecanismo para suministrar código a la herramienta se describe a continuación.

7.3.1. Definición de la función de *fitness*.

Para poder ejecutar un experimento, el usuario debe proporcionar el código Java que implementa la función de *fitness*. Para ello, debe codificar una clase que contenga un método que calcule el valor de *fitness* de un individuo concreto. Este valor tiene que ser positivo, y además, el mejor individuo tiene el mayor valor. El nombre de la clase es libre, es decir, lo determina el usuario, sin embargo, el método que implementa la función de *fitness* debe ser público y estático, y ajustado al siguiente prototipo:

double fitness (Fenotipo fenotipo)

En el proceso evolutivo, cada vez que es necesario evaluar el *fitness* de un individuo se invoca al método proporcionado por el usuario.

Toda la información que el usuario de **GEEMMA** necesita conocer para poder trabajar con la clase *Fenotipo* y el resto de las clases se puede encontrar en la documentación que se adjunta a la aplicación.

Por último, el nombre de la clase que contiene el método *fitness* es uno de los parámetros incluidos en la configuración de experimentos.

7.3.2. Definición de la semántica

La definición de la semántica se realiza también mediante una clase Java. Esta clase debe incorporar un método público y estático, y ajustado al siguiente prototipo:

```
public static boolean accionSemantica ( ArbolAnalisis arbol,
                                        Integer atributosSintetizados,
                                        Integer atributosHeredadosPadre,
                                        Integer atributosHeredadosHermanos,
                                        Regla regla)
```

En el proceso evolutivo, en concreto, en la construcción de los fenotipos de los individuos, el método *accionSemantica* se invoca en los momentos en los que se evalúan atributos. En el capítulo 5 se puede consultar los instantes en los que se realiza cálculo de atributos.

Queda fuera del alcance de este documento describir con detalle las recomendaciones para la implementación del método *accionSemantica* que, por otra parte, se pueden encontrar en la documentación que se adjunta a la aplicación.

7.4. Realización de experimentos

Como ya se indicó anteriormente, la aplicación permite ejecutar experimentos de manera individual o bien baterías de experimentos. En cualquiera de los dos casos, es necesario configurar el experimento antes de su ejecución. En la figura 7.1 aparece desplegado el menú a través del cual se configuran

y se ejecutan experimentos.

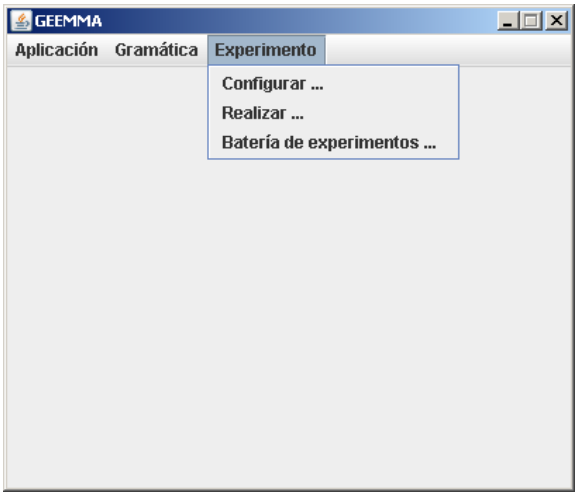

Figura 7.1: Menú de configuración y ejecución de experimentos.

7.4.1. Configuración de experimentos

El usuario puede configurar un experimento a través de la opción *Configurar ...* del menú de gestión de experimentos. Independientemente de si el usuario realiza un experimento individual o una batería de experimentos, la configuración siempre se realiza a través de la opción *Configurar ...* . Los parámetros configurables y su significado son los siguientes:

- **Semilla de números aleatorios**: valor que se utiliza para inicializar el generador de números aleatorios. Esta inicialización proporciona el mecanismo para la reproducción de experimentos.
- **Número de individuos de la población**: el tamaño de la población.
- **Número máximo de generaciones**: cota superior del número de generaciones que se utiliza para interrumpir el proceso evolutivo en el caso en el que no se encuentre una solución considerada aceptable.
- **Mínimo número de codones**: mínimo número de codones del genotipo de los individuos de la población utilizado en la inicialización aleatoria de la población.

- **Máximo número de codones**: máximo número de codones del genotipo de los individuos de la población utilizado en la inicialización aleatoria de la población.
- **Valor mínimo de un codón**: valor mínimo que puede tomar un codón.
- **Valor máximo de un codón**: valor máximo que puede tomar un codón.
- **Máximo *wrapping***: máximo número de aplicaciones del operador de *wrapping*. Si vale 0, el operador no se aplica.
- **Elitismo**: El algoritmo evolutivo de **GEEMMA** utiliza reemplazo generacional. Si se configura que sea *Elitista*, el mejor individuo de la población se mantiene de una generación a la siguiente, y si se configura *NO Elitista*, no se mantienen ningún individuo entre generaciones.
- **Tasa de recombinación**: Tanto por ciento de los individuos de la población que se reproducen.
- **Tasa de mutación**: Tanto por ciento de los individuos que mutan dentro del colectivo formado por los individuos generados a partir de la recombinación.
- **Nombre de la clase *fitness***: nombre de la clase Java que implementa el método para la evaluación del *fitness*.
- **Umbral de *fitness***: valor de *fitness* a partir del cual se considera que un individuo es una solución válida.
- **Fichero de la gramática independiente del contexto**: path del fichero que contienen la gramática independiente del contexto.
- **Método**: permite seleccionar uno de los dos métodos: Evolución Gramatical o Evolución Gramatical con Semántica.
- **Nombre de la clase semántica**: nombre de la clase Java que implementa el método que define la semántica del problema.

La ventana a través de la que se configura un experimento, y que se despliega tras la selección de la opción *Configurar ...* del menú de gestión de experimentos se muestra en la figura 7.2.

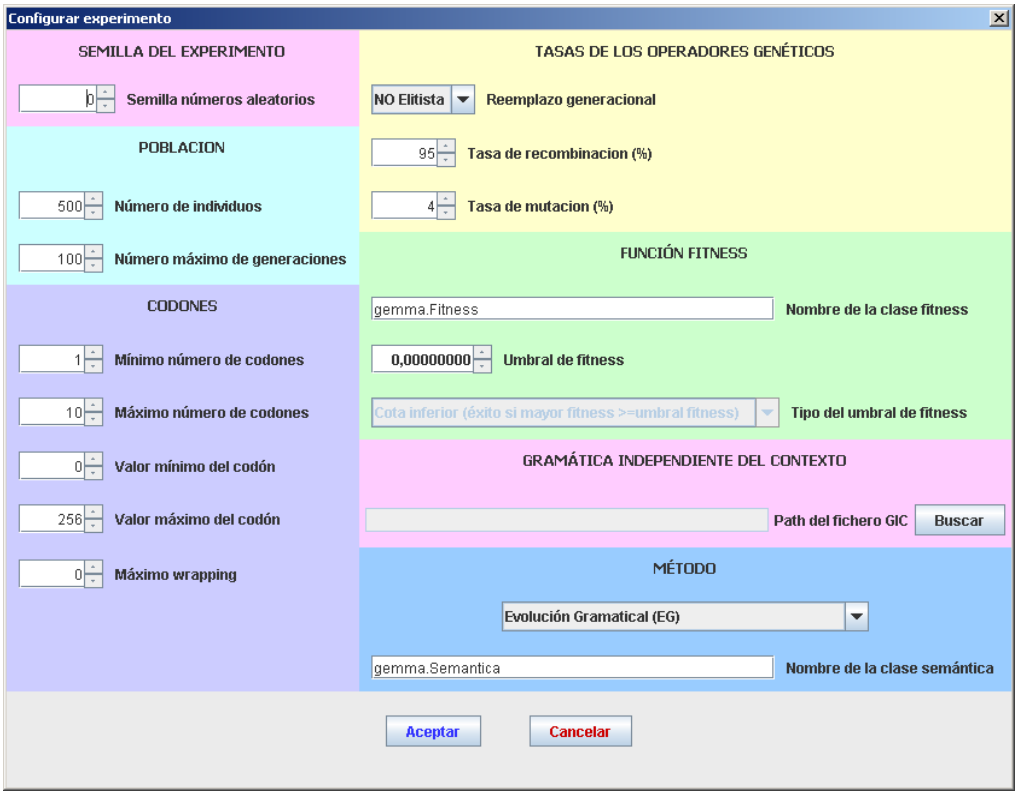

Figura 7.2: Ventana de configuración de experimentos.

7.4.2. Experimentos individuales

La realización de un experimento individual significa la aplicación del método seleccionado en la configuración del experimento en búsqueda de una solución. Para realizar un experimento individual, el usuario dispone de la opción *Realizar ...* del menú de gestión de experimentos. Esta opción sólo está operativa si el usuario ha configurado el experimento satisfactoriamente. Esto significa que, en la fase de configuración del experimento, la aplicación comprueba la validez de los parámetros, como por ejemplo, la existencia del fichero que contiene la gramática, el acceso a las clases Java proporcionadas por el usuario, etc, y si no son válidos, no se activa la opción de menú *Realizar ...* .

Una vez seleccionada la opción de realización de un experimento individual se despliega la ventana que se muestra en la figura 7.3.

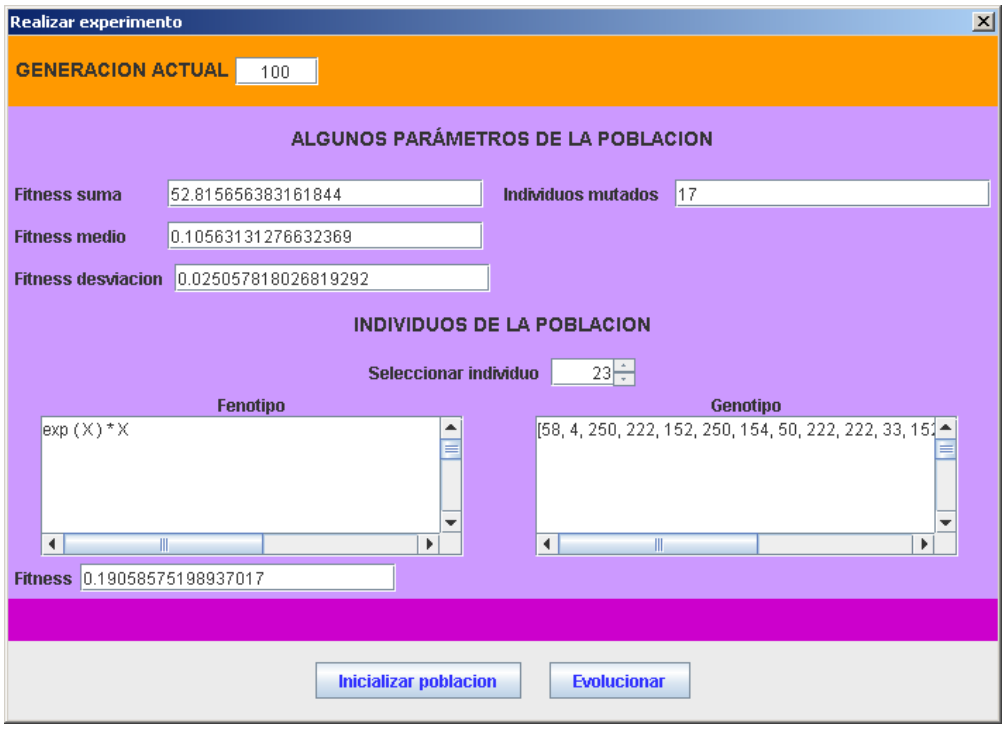

Figura 7.3: Ventana que muestra la evolución de un experimento individual.

Durante la ejecución de un experimento, se informa al usuario del estado

del mismo a través de los indicadores que aparecen en la ventana de la figura 7.3. Algunos de los indicadores son, por ejemplo, la generación en curso y el *fitness* medio de los individuos de la población. Además, una vez concluido el experimento, el usuario puede consultar el genotipo, el fenotipo y el *fitness* de cada uno de los individuos de la población.

7.4.3. Baterías de experimentos

Las baterías de experimentos permiten repetir un número determinado de veces el mismo experimento con distintas semillas. El acceso a esta funcionalidad se realiza a través de la opción *Batería de experimentos ...* del menú de gestión de experimentos. Una vez seleccionada esta opción, se despliega la ventana de la figura 7.4.

Figura 7.4: Ventana de configuración y evolución de una batería de experimentos.

Antes de realizar una batería de experimentos, el usuario debe configurar a través de los controles de la ventana de la figura 7.4 las semillas, el número de experimentos y el fichero de resultados.

El fichero de resultados de una batería de experimentos es la herramienta que permite realizar estimaciones acerca del rendimiento de los métodos, hacer estadísticas sobre la longitud de los genotipos de los mejores individuos, estudiar los fenotipos de los mejores individuos, etc. Es un fichero de texto que contiene, en primer lugar una cabecera informativa con los siguientes parámetros de configuración del experimento:

- Numero de individuos de la población.
- Mínimo numero de codones en la inicialización de la población.
- Máximo numero de codones en la inicialización de la población.
- Valor mínimo de codón.
- Valor maximo de codón.
- Máximo wrapping.
- Uso de elitismo.
- Tasa de recombinación.
- Tasa de mutación.
- Máximo numero de generaciones.
- Umbral de *fitness*.
- Nombre de la clase que implementa el método *fitness*.
- Fichero de la gramática independiente del contexto.
- Método.
- Nombre de la clase que implementa el método *accionSemantica*.

A continuación de la cabecera, el fichero de resultados contiene una línea de datos para cada experimento individual, con la siguiente información acerca de la ejecución del mismo:

- Número secuencial del experimento.

- Semilla con la que se ha inicializado el generador de números aleatorios.
- Generación en la que ha terminado el experimento.
- Número de individuos mutados durante el experimento.
- Número de individuos rechazados por ser semánticamente incorrectos.
- *Fitness* del mejor individuo de la última generación del experimento.
- Fenotipo del mejor individuo de la última generación del experimento.
- Genotipo del mejor individuo de la última generación del experimento.

7.5. Gestión de gramáticas independientes del contexto

GEEMMA proporciona una cómoda interfaz que le permite al usuario definir gramáticas independientes del contexto y su almacenamiento/recuperación en disco. En la figura 7.5 se muestra el menú disponible para todas las operaciones relacionadas con la gestión de gramáticas.

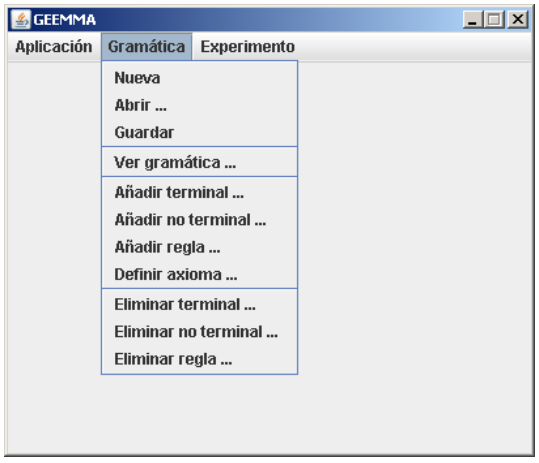

Figura 7.5: Menú de gestión de gramáticas independientes del contexto.

El menú desplegado en la figura 7.5 permite al usuario definir los cuatro componentes de la gramática: el conjunto de terminales, el conjunto de

no terminales, el axioma y el conjunto de reglas. El usuario también tiene la posibilidad de visualizar la gramática y almacenarla o recuperarla de disco.

La definición del conjunto de terminales se realiza a través de las opciones de menú *Añadir terminal ...* y *Eliminar terminal ...*, y los no terminales a través de *Añadir no terminal ...* y *Eliminar no terminal ...*.Para definir el axioma se dispone de la opción de menú *Definir axioma ...*. En la figura 7.6 se muestran las ventanas de diálogo que se despliegan tras la selección de una de las cinco anteriores opciones de menú. Dichas ventanas disponen de los elementos necesarios (cajas de texto, listas, botones) para que el usuario añada o elimine símbolos.

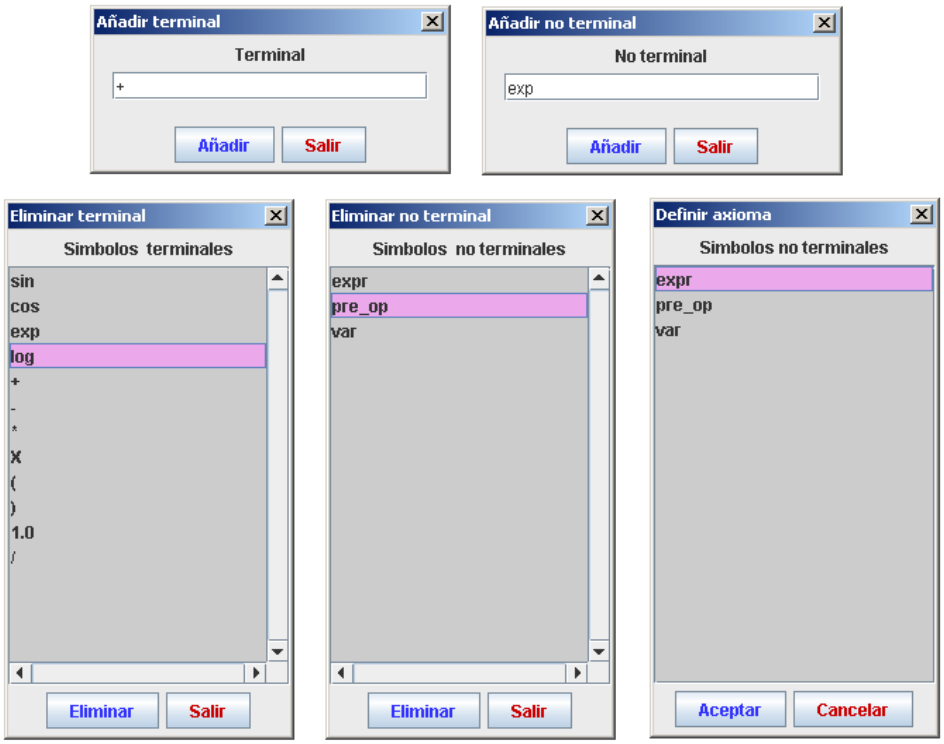

Figura 7.6: Ventanas de diálogo para la definición de los símbolos de la gramática.

La incorporación de una nueva regla a la gramática se hace a través de la opción de menú *Añadir regla ...*. Una vez seleccionada dicha opción,

el sistema despliega una ventana en la que el usuario dispone de los conjuntos de terminales y no terminales de la gramática para construir la nueva regla. En la figura 7.7 se muestra dicha ventana de edición de reglas.

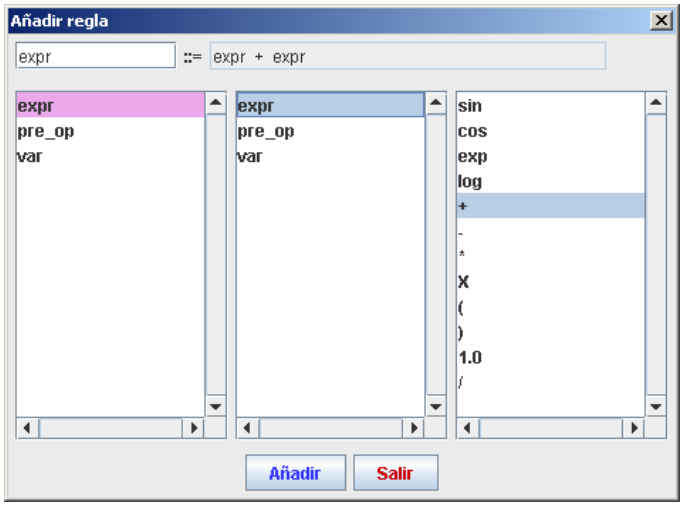

Figura 7.7: Ventana de edición de reglas.

Para eliminar una regla de la gramática, la opción *Eliminar regla ...* muestra la venta de la figura 7.8 en la que aparece una lista con todas las reglas de la gramática para que el usuario seleccione la regla que se va a eliminar.

Las operaciones de almacenamiento y recuperación de disco son accesibles a través de las opciones *Abrir ...* y *Guardar ....* La selección de cualquiera de las dos opciones hace que se despliegue una ventana de exploración del sistema de ficheros para que el usuario seleccione un fichero concreto.

La visualización de la gramática completa es accesible mediante la opción *Ver gramática ...* que muestra una ventana como la de la figura 7.9 en la que aparecen los conjuntos de terminales, no terminales, el axioma y las reglas de la gramática.

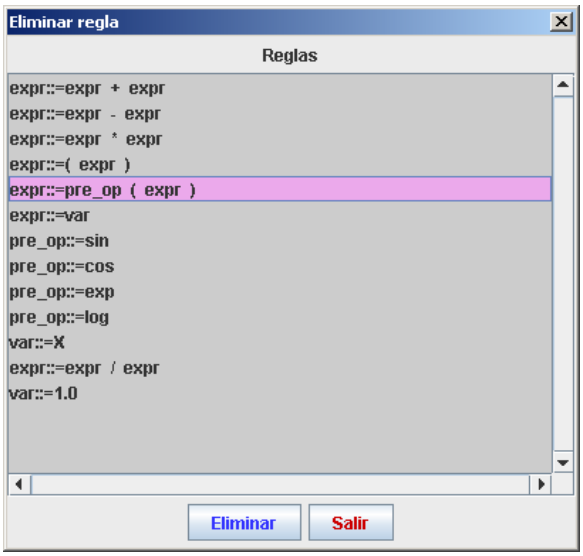

Figura 7.8: Ventana de diálogo para la eliminación de reglas.

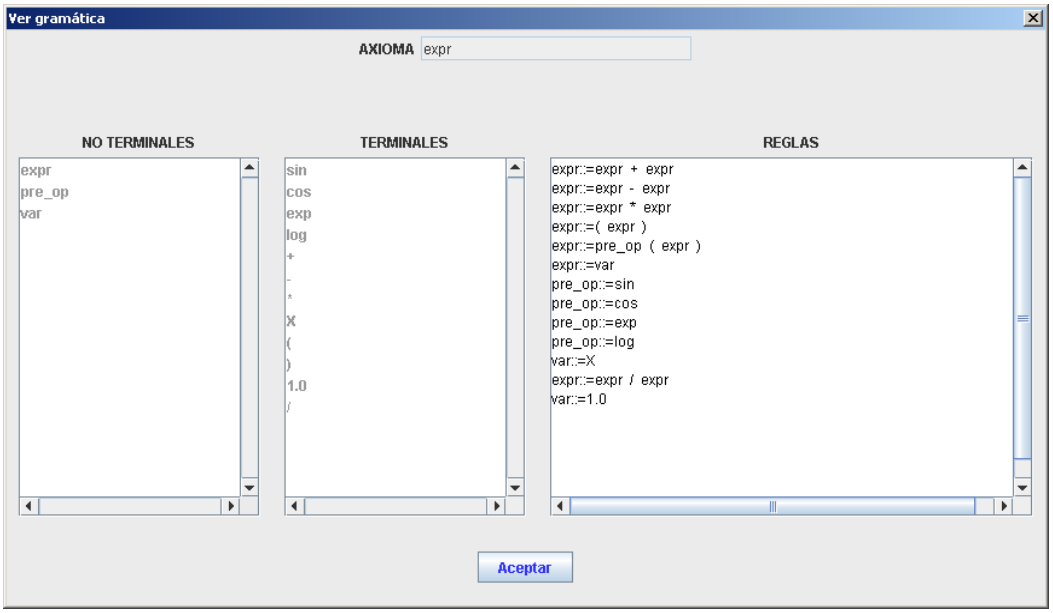

Figura 7.9: Ventana de información sobre los componentes de la gramática.

Capítulo 8

Conclusiones y líneas futuras de investigación

En este capítulo se describen las conclusiones del trabajo desarrollado en esta tesis así como las líneas futuras de investigación.

8.1. Conclusiones

En esta tesis se han presentado dos nuevos métodos de programación genética lineal que incorporan a mecanismos ya existentes de representación de restricciones sintácticas (construcciones independientes del contexto) la posibilidad de describir restricciones de las consideradas en esta tesis como "semánticas" (no expresables con construcciones independientes del contexto) de forma que sólo pertenecen a la población sujeta al proceso evolutivo individuos que satisfacen todas las restricciones. Los dos métodos presentados en esta tesis son extensiones de otro sistema de programación genética lineal llamado Evolución Gramatical (EG). EG incorpora las restricciones sintácticas mediante el uso explícito de una gramática independiente del contexto que puede generar los individuos (fenotipos) buscados.

La primera extensión, Evolución Gramatical con Gramáticas de Atributos (EGA) incorpora la semántica haciendo uso de gramáticas de atributos en lugar de gramáticas independientes del contexto. Las gramáticas de atributos son una bien conocida herramienta utilizada de forma exhaustiva en el área de la informática dedicada a los procesadores de lenguaje (intérpretes y compiladores). La razón de su éxito puede encontrarse en su estructura: mantienen un núcleo independiente del contexto (realmente una gramática independiente del contexto) a la que añaden un sistema de atributos para poder especificar condiciones no representables (o representables de una ma-

nera menos cómoda) mediante construcciones independientes del contexto. El núcleo independiente del contexto asegura la existencia de algoritmos de análisis eficientes y fáciles de programar.

En la segunda extensión, Evolución Gramatical con Gramáticas de Christiansen (EGC), se hace uso de las gramáticas de Christiansen para la incorporación de la semántica. Las gramáticas de Christiansen son una extensión adaptable de las gramáticas de atributos. La propiedad de adaptabilidad (la capacidad de modificar la gramática a medida que se utiliza) más que potencia expresiva hace que ciertas restricciones sean más cómodamente especificables con herramientas adaptables que con no adaptables (como las gramáticas de atributos).

A pesar de su juventud (algo más de una década) ha habido una actividad bastante intensa en el estudio y posibles extensiones a EG. Tanto para EG como para otras técnicas de programación genética, siempre se ha mostrado interés por el posible uso de representaciones formales (como las gramáticas de la jerarquía de Chomsky y otros modelos similares) para atacar el problema del cierre sintáctico así como por la posibilidad de encontrar formas de incorporar también restricciones "semánticas" (no expresables con construcciones independientes del contexto). Los intentos anteriores en este sentido han resultado particulares, poco generales, incompletos o parcialmente especificados o desarrollados. Esta tesis, por tanto, encuentra todo su sentido en este contexto y en este problema abierto.

Además de las propuestas teóricas, se ha desarrollado la aplicación **GEEMMA** con el objetivo de disponer de un entorno en el que probar los métodos teóricos propuestos y así poder validar su funcionamiento y evaluar su rendimiento. Se han realizado experimentos de regresión e integración simbólica de funciones reales, aplicando EG y EGA y se han comparado los resultados obtenidos. También se han hecho experimentos de regresión lógica, con restricciones relativas a la pertenencia de los operadores al mismo conjunto completo, aplicando EG y EGC y se han comparado los resultados obtenidos.

Las principales conclusiones a las que se ha llegado son las siguientes:

- En primer lugar, cuando se propone la extensión de un método, como es el caso de esta tesis, y se pretende evaluar su rendimiento, y compararlo con el método extendido, surge la necesidad de desarrollar una herramienta que implemente ambos métodos. Esta circunstancia se evitaría si existiera una base de datos para intercomparación de resultados de métodos en el área de la programación genética, como existe, por

ejemplo en el área de las redes neuronales. Además, incluso implementando el método con el que se quiere comparar, la reproducción de los experimentos que proponen los autores del método es difícil, porque habitualmente no se describen de forma completa los parámetros involucrados en cada experimento. Por lo tanto, se concluye que sería muy interesante la creación de una base de datos orientada a la comparación de sistemas de programación genética.

- En segundo lugar, en cuanto a la comparación de los nuevos métodos (EGA y EGC) que extienden EG es importante destacar que no es fácil encontrar un problema en el que se puedan comparar, en términos de rendimiento, los resultados obtenidos. Por una parte, si las restricciones del problema son completamente expresables mediante una gramática independiente del contexto, parece razonable pensar que EGA y EGC son métodos con una capacidad expresiva que excede a las necesidades del problema. Por otra parte, si el problema tiene unas restricciones que no se pueden expresar únicamente con una gramática independiente del contexto, entonces, simplemente, no se puede abordar de forma natural con EG aunque, en algunos casos, sea posible incorporar ciertas restricciones a través de la función de *fitness*. Las pruebas realizadas para el problema de regresión simbólica de funciones reales muestran que el rendimiento obtenido con EGA es superior al obtenido con EG en la mayoría de los experimentos, llegando a mejoras de más del 30 % en la probabilidad acumulada de éxito cuando se incorporan al sistema las restricciones semánticas menos exigentes. Además, esta mejoría se ve incrementada si se endurecen las restricciones semánticas. Los resultados obtenidos para el problema de integración simbólica muestran de nuevo que el método EGA proporciona mejores rendimientos que EG, y que el endurecimiento de las restricciones semánticas incrementa los niveles de rendimiento. El problema de regresión lógica con restricciones de los operadores resuelto con EGC permite concluir que, los resultados obtenidos con EGC son significativamente superiores en términos de rendimiento (con mejoras del 80 % en la probabilidad acumulada de éxito en la generación 100) cuando en EG se incorporan las restricciones de los operadores en la función de *fitness*. Sin embargo, si en EG se utiliza una gramática independiente del contexto que exprese las restricciones de los operadores, entonces, los rendimientos obtenidos con EG y EGC son comparables. Este resultado es razonable ya que, en ambos métodos, todos los fenotipos cumplen la restricción relativa a la mezcla de operadores, aunque el mecanismo para obtener dichos fenotipos sea diferente, y por lo tanto la calidad de la población es

similar en ambos métodos.

Las técnicas propuestas en esta tesis no deben ser sólo evaluadas desde el punto de vista del rendimiento sino también desde la perspectiva de lo que hemos denominado "confort de diseño" a lo largo del texto. Es evidente que los métodos EGA y EGC propuestos en esta tesis utilizan formalismos más complejos que las gramáticas independientes del contexto del método EG. Esta complejidad obliga al usuario de los métodos a conocer los formalismos, pero, una vez adquirido dicho conocimiento, existe un conjunto de problemas para los que los métodos EGA y EGC permitan diseñar soluciones de manera más directa y natural. Como ejemplo de este "confort de diseño" podemos mencionar el problema de regresión lógica con restricciones sobre los operadores. La solución a dicho problema utilizando EG con una gramática independiente del contexto que permita la mezcla de operadores y una función de *fitness* que incluya la comprobación de las restricciones, dibuja un escenario en el que, se generan soluciones tanto válidas como inválidas en cuanto a las restricciones y, posteriormente, se comprueban en la función de *fitness* que evidentemente tiene un nivel de complejidad superior al que corresponde a la simple evaluación de la función en una serie de puntos. Este escenario parece menos directo y natural que el que proporciona la solución con EGC, en el que, simplemente, ante la primera aparición de un operador, se imposibilita el uso de los operadores que no pertenezcan a su conjunto, generando siempre soluciones válidas, y utilizando una sencilla función de *fitness* que únicamente implica la evaluación de la solución.

- Otro de los temas de interés del grupo de investigación al que pertenezco son los nuevos modelos de cómputo, en especial los bioinspirados, incluidos en lo que se está comenzando a llamar *Computación Natural*. La programación genética *tradicional* está orientada a lenguajes de programación para *arquitecturas clásicas* esencialmente basadas en la de von Neumann. En la actualidad está resurgiendo el interés por modelos bioinspirados bien conocidos (autómatas celulares, gramáticas de derivación paralela o sistemas de Lindenmayer) coetáneos de otros modelos bioinspirados que por su aplicación, naturaleza y desarrollo científico se han convertido ya en paradigmas propios (como la computación evolutiva, las redes neuronales, o los sistemas multiagente). Este interés se ha concretado en la definición y estudio como nuevas arquitecturas de computación de nuevos modelos inspirados en la naturaleza (sistemas de gramáticas, computación basada en ADN y en membranas o sistemas P, redes de procesadores que evolucionan) Todas tienen en común

la máxima potencia expresiva (equivalentes a la máquina de Turing) una naturaleza intrínsecamente paralela en la que se pueden definir algoritmos para problemas costosos para las arquitecturas clásicas que mejoren su rendimiento temporal y la casi inexistencia de arquitecturas físicas que los implementen. Sin embargo se ha demostrado que se pueden diseñar lenguajes de programación para ellos y se pueden utilizar para solucionar problemas concretos. No existe nada parecido a lo que llamamos *Ingeniería de Software* por lo que resulta muy atractiva la posibilidad de diseñar una metodología de programación automática para estos modelos. El trabajo, en aquellos para los que no existen todavía lenguajes de programación, implica diseñar una representación formal (gramatical) que genere casos particulares y simuladores que puedan ejecutarlos para poder definir funciones de adecuación que nos indiquen cuáles de esos casos particulares resuelven mejor el problema bajo consideración. En esta tesis se muestran también diferentes resultados en esta línea. En particular se muestra gramáticas de Christiansen capaces de generar casos particulares de sistemas H (inspirados en ADN) y sisemas P (en membranas). Otro trabajo relacionado muestra cómo se puede atacar el diseño automático de modelos matemáticos para un problema concreto en el ámbito de la psicología.

- La última y más rotunda conclusión es que queda mucho trabajo por hacer, lo que nos permite mantener actividad en nuestra línea de investigación como se describe en el siguiente apartado.

8.2. Líneas futuras de investigación

En cuanto a futuras líneas de investigación nos planteamos:

- Ampliar **GEEMMA** para tener una interfaz gráfica para la edición de la función de *fitness* y para la especificación de la semántica. En ese sentido, esta tesis enlaza con otra de las líneas de interés del grupo al que pertenezco y en el que estoy colaborando con otros investigadores tanto de nuestro grupo de investigación en la UAM como de la URJC y de la UPM. Consideramos las gramáticas de atributos y las de Christiansen realmente como lenguajes de programación, por lo tanto, estamos interesados en la implantación de herramientas de ayuda al desarrollo y la programación con ellos. Estas herramientas tienen necesariamente que incorporar tanto facilidades de edición como de depuración. Esto implica abordar de una manera sistemática, general y práctica el problema de la especificación de dependencias circulares entre los atributos. Las

gramáticas de atributos son un modelo ampliamente estudiado. Exceptuándonos a nosotros, el uso de las de Christiansen se ha restringido casi por completo al ámbito de la programación lógica. Las herramientas que describimos en esta línea abierta tendrán que abordar problemas más novedosos en los modelos adaptables como los algoritmos de análisis.

- Uno de los grandes inconvenientes de enfoques como el nuestro es el del incremento en el tiempo de tratamiento de cada individuo. Somos conscientes de que uno de los precios de generalizar la incorporación de semántica y las representaciones formales de los individuos es éste. Por lo tanto, nos resulta de mucho interés acceder a plataformas masivamente paralelas (como los clústers de ordenadores) que nos permitan paliar este inconveniente. En ese sentido estoy también en contacto con otros investigadores de nuestro grupo que están desarrollando plataformas de propósito general en las que paralelizar algoritmos secuenciales para ser ejecutados en plataformas paralelas optimizando el balanceado de carga en los diferentes nodos y minimizando el tiempo de comunicación entre ellos. Su uso implica implementar una herramienta similar a **GEEMMA** compatible con esta plataforma. En este caso implica el uso de C++ y MPI.

- Aplicar esta misma forma de representar restricciones sintácticas y semánticas a otras técnicas de búsqueda de propósito general (como Swarm Particle Optimization o su variante gramatical Grammatical Swarm)

- Completar el estudio sistemático del rendimiento de las propuestas con problemas nuevos.

- Uno de los puntos en los que más creemos que se puede avanzar es en el estudio en profundidad de los operadores genéticos y en la propuesta de nuevas variantes de ellos que casi con seguridad implicarán formas más sofisticadas de traducción de genotipo a fenotipo. En ese sentido estamos en contacto con otros investigadores de la Universidad de Cantabria que ya han tenido éxito en la propuesta de variantes a la programación genética que hibridan la experiencia de la programación evolutiva con la del aprendizaje automático, la minería de datos y las estructuras de datos de la teoría de la complejidad algebraica. Creemos que será de mucho interés plantear nuevos métodos que incorporen las ventajas de estos y que añadan también la potencia de las representaciones formales para especificar restricciones sintácticas y semánticas.

- Respecto a la programación de nuevos modelos de cómputo, estoy en contacto con otros investigadores de nuestro grupo y de otros de la Universidad Rovira i Virgili y de Sevilla para desarrollar herramientas de programación automática en computación natural y comparar los resultados obtenidos con los desarrollados "a mano" para evaluar la viabilidad de una posible metodología general de programación automática genética para estos modelos de cómputo.

Bibliografía

Aho, A. V.; Lam, M. S.; Sethi, R. y Ullman, J. D. (2008). *Compiladores: principios, técnicas y herramientas*. Pearson Educación, 2ª edición.

Alfonseca, M.; de la Cruz, M.; Ortega, A. y Pulido, E. (2007). *Compiladores e intérpretes. Teoría y práctica*. Pearson.

Alonso, C.L.; Puente, J. y Montaña, J.L. (2008). «Straight Line Programs: A New Linear Genetic Programming Approach». En: *20th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2008), November 3-5, 2008, Dayton, Ohio, USA, Volume*, pp. 517–524. IEEE Computer Society.

Banzhaf, W. (1994). «Genotype-Phenotype-Mapping and Neutral Variation - A Case Study in Genetic Programming». En: *PPSN III: Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature*, pp. 322–332. Springer-Verlag, London, UK.

Banzhaf, W.; Nordin, P.; Keller, R. y Francone, F (1998). *Genetic Programming: An Introduction*. Artificial Intelligence. Morgan Kaufmann.

Beadle, L. y Johnson, C. (2008). «Semantically driven crossover in genetic programming». En: *IEEE Congress on Evolutionary Computation*, pp. 111–116. IEEE.

Beadle, L. y Johnson, C. (2009). «Semantically driven mutation in genetic programming». En: *IEEE Congress on Evolutionary Computation*, pp. 1336–1342. IEEE.

Brameier, M. y Banzhaf, W. (2001). «A comparision of linear genetic programming and neural networks in medical data mining». *IEEE Transactions on Evolutionary Computation*, **5(1)**, pp. 17–26.

Bratko, I. (1990). *Prolog Programming for Artificial Intelligence*. Addison-Wesley Publishers Company.

Chomsky, N. (1956). «Three models for the description of language». *Transactions on information theory*, **2(2)**, pp. 113–167.

Chomsky, N. (1959). «On certain formal properties of grammars». *Information and control*, **2(2)**, pp. 137–167.

Christiansen, H. (1985). «Syntax, semantics, and implementation strategies for programming languages with powerful abstraction mechanisms». volumen 1 de *Datalogiske Skrifter*. Roskilde University, Roskilde, Denmark.

Christiansen, H. (1986). «Parsing and compilation of generative languages». volumen 3 de *Datalogiske Skrifter*. Roskilde University, Roskilde, Denmark.

Christiansen, H. (1988a). «Programming as language development». volumen 15 de *Datalogiske Skrifter*. Roskilde University, Roskilde, Denmark.

Christiansen, H. (1988b). «The syntax and semantics of extensible languages». volumen 14 de *Datalogiske Skrifter*. Roskilde University, Roskilde, Denmark.

Christiansen, H. (1990). «A survey of adaptable grammars». *SIGPLAN Notices*, **25(11)**, pp. 35–44.

Christiansen, H. (2009). «Adaptable grammars for non-context-sensitive languages». En: J. Cabestany Moncusi; F. Sandoval; A. Prieto y J.M. Corchado (Eds.), *10th Int. Conf. on Artificial Neural Networks, IWANN'09*, volumen 5517 de *Lecture Notes in Computer Science*, pp. 488–495. Springer, Berlin / Heidelberg.

Cleary, R. y O'Neill, M (2005). «An Attribute Grammar Decoder for the 01 MultiConstrained Knapsack Problem». En: Günther R. Raidl y Jens Gottlieb (Eds.), *Evolutionary Computation in Combinatorial Optimization, 5th European Conference, EvoCOP 2005, Lausanne, Switzerland, March 30 - April 1, 2005, Proceedings*, volumen 3448 de *Lecture Notes in Computer Science*, pp. 34–45. Springer.

Cramer, N. L. (1985). «A Representation for the Adaptive Generation of Simple Sequential Programs». En: *Proceedings of the 1st International Conference on Genetic Algorithms*, pp. 183–187. L. Erlbaum Associates Inc., Hillsdale, NJ, USA.

Crepeau, R. (1995). «Genetic Evolution of Machine Language Software». En: Justinian P. Rosca (Ed.), *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pp. 121–134. Tahoe City, California, USA.

de la Cruz, M.; Ortega, A. y Alfonseca, M. (2005). «Attribute Grammar Evolution». En: J. Mira y J.R. Álvarez (Eds.), *Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach*, volumen 3562 de *Lecture Notes in Computer Science*, pp. 182–191. Springer, Berlin / Heidelberg.

Dembinski, P. y Maluszynski, J. (1978). «Attribute Grammars and Two-Level Grammars: A Unifying Approach». En: Józef Winkowski (Ed.), *MFCS*, volumen 64 de *Lecture Notes in Computer Science*, pp. 143–154.

Deransart, P. y Maluszynski, J. (1985). «Relating Logic Programs and Attribute Grammars». *Journal of Logic Programming*, **2(2)**, pp. 119–155.

di Forino, A. C. (1963). «Some remarks on the syntax of symbolic programming languages». *Communications of the ACM*, **6(8)**, pp. 456–460.

Eiben, A. E. y Smith, J. E. (2003). *Introduction to Evolutionary Computing*. Springer.

Freeman, J.J. (1998). «A Linear Representation for GP using Context Free Grammars». En: J. Koza; W. Banzhaf; K. Chellapilla; K. Deb; M. Dorigo; D. Fogel; M. Garzon; D. Goldberg; H. Iba y R. Riolo (Eds.), *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pp. 72–77. Morgan Kaufmann, University of Wisconsin, Madison, Wisconsin, USA.

Gordon, M.J.C. (1979). *The Denotational Description of Programming Languages: An Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Gruau, F. (1994). *Neural Network Synthesis Using Cellular Encoding And The Genetic Algorithm*. Tesis doctoral, Ecole Normale Supirieure de Lyon, France.

Grune, D.; Bal, H.E.; Jacobs, J. H. y Langendoen, K.G. (2007). *Diseño de compiladores modernos*. McGraw-Hill.

Hörner, H. (1996). «A C++ Class Library for Genetic Programming: The Vienna University of Economics Genetic Programming Kernel.». Vienna University of Economics.

Johnson, C. (2002a). «Deriving genetic programming fitness properties by static analysis». En: James A. Foster; Evelyne Lutton; Julian Miller; Conor Ryan y Andrea G. B. Tettamanzi (Eds.), *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volumen 2278 de *LNCS*, pp. 298–307. Springer-Verlag.

Johnson, C. (2002b). «What can automatic programming learn from theoretical computer science?» En: Xin Yao (Ed.), *Proceedings of the 2002 UK Workshop on Computational Intelligence*, University of Birmingham.

Johnson, C. (2007). «Genetic Programming with Fitness based on Model Checking». En: Marc Ebner; Michael O'Neill; Anikó Ekárt; Leonardo Vanneschi y Anna Isabel Esparcia-Alcázar (Eds.), *Proceedings of the 10th European Conference on Genetic Programming*, volumen 4445 de *Lecture Notes in Computer Science*, pp. 114–124. Springer.

Keller, R.E. y Banzhaf, W. (1996). «Genetic programming using genotype-phenotype mapping from linear genomes into linear phenotypes.» En: Koza.J.R.; D.E. Goldberg; D.B. Fogel y Riolo R.L. (Eds.), *Genetic Programming 1996 : Proceedings Of The First Annual Conference*, pp. 116–122. MIT Press, Stanford University, CA, USA.

Kimura, M. (1983). *The neutral theory of molecular evolution*. Cambridge University Press.

Knuth, D. (1968). «Semantics of Context-Free Languages». *Mathematical Systems Theory*, **2(2)**, pp. 127–145.

Knuth, D. (1971). «Correction: Semantics of Context-Free Languages». *Mathematical Systems Theory*, **5(1)**, pp. 95–96.

Koza, J. (1992). *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge.

Koza, J. (1994). *Genetic programming II: automatic discovery of reusable programas*. MIT Press, Cambridge.

Koza, J. (1999). *Genetic programming III: darwiniang invention and problem solving*. Morgan Kaufmann.

Koza, J. (2005). *Genetic programming IV: Routine Human-competitive Machine Intelligence*. Springer.

Levine, J.R.; Mason, T. y Brown, D. (1992). *lex & yacc*. O'Reilly, 2ª edición.

Lewin, B. (1999). *Genes VII.* Oxford University Press.

Majeed, H. y Ryan, C. (2006a). «A Less Destructive, Context-Aware Crossover Operator for GP». En: *Genetic Programming, 9th European Conference, EuroGP 2006, Budapest, Hungary, April 10-12, 2006, Proceedings*, volumen 3905 de *Lecture Notes in Computer Science*, pp. 36–48. Springer.

Majeed, H. y Ryan, C. (2006b). «Using context-aware crossover to improve the performance of GP». En: Mike Cattolico (Ed.), *Genetic and Evolutionary Computation Conference, GECCO 2006, Proceedings, Seattle, Washington, USA, July 8-12, 2006*, pp. 847–854. ACM.

Majeed, H. y Ryan, C. (2007). «Context-aware mutation: a modular, context aware mutation operator for genetic programming». En: *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pp. 1651–1658. ACM, New York, NY, USA.

Marcotty, H., M.and Ledgard y Bochmann, G.V. (1976). «A Sampler of Formal Definitions». *ACM Comput. Surv.*, **8(2)**, pp. 191–276.

Mason, K. P. (1987). «Dynamic template translators - a new device for specifying programming languages». *International Journal of Computer Mathematics*, **22(3)**, pp. 199–212.

Mason, K.P. (1984). *Dynamic Template Translators: A useful model for the definition of programming languages.* Tesis doctoral, University of Adelaide, Australia.

Meek, B. (1990). «The static semantics file». *SIGPLAN Not.*, **25(4)**, pp. 33–42.

Meyer, B. (1990). *Introduction to the theory of programming languages.* Prentice Hall.

Nielson, H.R. y Nielson, F. (1992). *Semantics with applications: a formal introduction.* John Wiley & Sons.

Nilsson, I. y Maluszynski, J. (1990). *Logic, programming and prolog.* John-Wiley & sons.

Nordin, P. (1994). *Advances in genetic programming 1.* capítulo A compiling genetic programming system that directly manipulates de machine-code, pp. 311–331. MIT Press.

Nordin, P. (1997). *Evolutionary program induction of binary machine code and its applications*. Tesis doctoral.

Nordin, P.; Banzhaf, W. y Francone, F. (1999). *Advances y genetic programming 3*. capítulo Efficient Evolution of Machine Code for CISC Architectures using Blocks and Homologous Crossover, pp. 275–299. MIT Press.

O'Neill, C., M.and Ryan (2001). «Grammatical Evolution». *IEEE Transactions on Evolutioray Computation*, **5**, pp. 349–358.

O'Neill, M.; Cleary, R. y Nikolov, N. (2004). «Solving Knapsak Problems with Attribute Grammars». En: R. Poli; S. Cagnoni; M. Keijzer; E. Costa; F. Pereira; G. Raidl; S. C. Upton; D. Goldberg; H. Lipson; E. de Jong; J. Koza; H. Suzuki; H. Sawai; I. Parmee; M. Pelikan; K. Sastry; D. Thierens; W. Stolzmann; P. L. Lanzi; S. W. Wilson; M. O'Neill; C. Ryan; T. Yu; J. F. Miller; I. Garibay; G. Holifield; A. S. Wu; T. Riopka; M. M. Meysenburg; A. W. Wright; N. Richter; J. H. Moore; M. D. Ritchie; L. Davis; R. Roy y M. Jakiela (Eds.), *GECCO 2004 Workshop Proceedings*, Seattle, Washington, USA.

O'Neill, M. y Ryan, C. (2003). *Grammatical Evolution. Evolutionary Automatic Programming in an Arbitrary Languaje*. Kluwer Academic Publishers.

Ortega, A.; de la Cruz, M. y Alfonseca, M. (2007). «Christiansen Grammar Evolution: Grammatical Evolution With Semantics». *IEEE Transactions on Evolutionary Computation*, **11(1)**, pp. 77–90.

O'Sullivan, J. y Ryan, C. (2002). «An Investigation into the Use of Different Search Strategies with Grammatical Evolution». En: *Genetic Programming*, volumen 2278 de *Lecture Notes in Computer Science*, pp. 268–277. Springer, Berlin / Heidelberg.

Paterson, N. (2002). *Genetic programming with context-sensitive grammars*. Tesis doctoral, Saint Andrew's University.

Paterson, N. y Livesey, M. (1996). «Distinguishing genotype and phenotype in genetic grogramming». En: R. Koza; D. Goldberg; D. Fogel y R. Riolo (Eds.), *Genetic Programming 1996: Proceedings of the First Annual Conference*, pp. 141–150. MIT Press.

Pereira, F. C. N. y Warren, D. H. D. (1980). «Definite clause grammars for language analysis–A survey of the formalism and a comparison with

augmented transition networks». *Artificial Intelligence*, **13(3)**, pp. 231 – 278.

Prechelt, L. (1994). «PROBEN1 - A Set of Neural Network Benchmark Problems and Benchmarking Rules». *Informe técnico*, Univ. Karlsruhe, Germany.

Ryan, C.; Collins, J.J. y O'Neill, M. (1998). «Grammatical Evolution: Evolving Programs for an Arbitrary Language». En: *Genetic Programming. First European Workshop, EuroGP'98 Paris, France, April 14-15, 1998 Proceedings*, volumen 1391 de *Lecture Notes in Computer Science*, pp. 83–95. Springer, Berlin / Heidelberg.

Shutt, John N. (1993). *Recursive Adaptable Grammars*. Tesina o Proyecto, Faculty of the Worcester Polytechnic Institute.

Slonneger, K. y Kurtz, B. (1995). *Formal syntax and semantics of programming languages: a lboratory based approach*. Addison-Wesley Publishers Company.

Teller, A. y Veloso, M. (1995a). «A Controlled Experiment: Evolution for Learning Difficult Image Classification». En: *Seventh Portuguese Conference On Artificial Intelligence*, volumen 990 de *Lecture Notes in Computer Science*, pp. 165–176. Springer-Verlag, Funchal, Madeira Island, Portugal.

Teller, A. y Veloso, M. (1995b). «PADO: Learning Tree Structured Algorithms for Orchestration into an Object Recognition System». *Informe técnico CMU-CS-95-101*, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA.

Uy, N. Q.; Hoai, N. X y O'Neill, M. (2009a). «Semantic Aware Crossover for Genetic Programming: The Case for Real-Valued Function Regression». En: Leonardo Vanneschi; Steven Gustafson; Alberto Moraglio; Ivanoe De Falco y Marc Ebner (Eds.), *Genetic Programming, 12th European Conference, EuroGP 2009, Tübingen, Germany, April 15-17, 2009, Proceedings*, volumen 5481 de *Lecture Notes in Computer Science*, pp. 292–302. Springer.

Uy, N. Q.; Hoai, N. X; O'Neill, M.; McKey, B. y Galván-López, E. (2009b). «An Analysis of Semantic Aware Crossover». En: Leonardo Vanneschi; Steven Gustafson; Alberto Moraglio; Ivanoe De Falco y Marc

Ebner (Eds.), *Computational Intelligence and Intelligent Systems*, volumen 51 de *Communications in Computer and Information Science*, pp. 56–65. Springer-Verlag.

van Wijngaarden, A. (1965). «Orthogonal Design and Description of a Formal Language». *Informe técnico 76*, Mathematisch Centrum, Amsterdam.

van Wijngaarden, A. (1977). «Revised report on the algorithmic language ALGOL 68». *SIGPLAN Notices*, **12(5)**, pp. 1–70.

Wegbreit, B. (1970). *Studies in Extensible Programming Languages*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York.

Whigham, P.A. (1996). *Grammatical bias for evolutionary learning*. Tesis doctoral, New South Wales, Australia, Australia.

Yu, T. y Bentley, P. (1998). «Methods to evolve legal phenotypes». En: A.E. Eiben; T. Back; M Schoenauer y H. P. Schwefel (Eds.), *Fifth International Conference on Parallel Problem Solving from Nature*, volumen 1498 de *Lecture Notes in Computer Science*, pp. 280–291. Springer, Amsterdam.